Pedro Miguel Pereira Serrano Martins

Licenciado em Engenharia Informática

Evaluation and Optimization of a Session-based Middleware for Data Management

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientadora : Maria Cecília Gomes, Prof[a]. Auxiliar, Universidade Nova de Lisboa

Co-orientador : Hervé Paulino, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Carlos Augusto Issac Piló Viegas Damásio

Arguente: Manuel Martins Barata

Vogal: Cecília Gomes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

June, 2014

Evaluation and Optimization of a Session-based Middleware for Data Management

I dedicate this thesis to my new life, the one I have been fighting for over 24 years now, and that will begin with the end of this thesis. I sure hope it is worth all the trouble. Oh, and also to my family.

Acknowledgements

I want to thank my bosses, Prof. Hervé Paulino and Maria Cecília for all the help and the late, late, late nights of hard work put into this thesis. I also want to thank to my mother and father for their unrelenting support, even in the darkest times.

Furthermore, I wish to extend my thanks to the various communities that helped me. This would not have been possible without them:

- http://stackoverflow.com/
- http://tex.stackexchange.com/
- Camel mailing list
- Esper mailing list

And last, but not least, a special thanks and big hugs to the AWS support team, specially to miss Ashley L., who guarantied the future of the project through the AWS system.

Abstract

The current massive daily production of data has created a non-precedent opportunity for information extraction in many domains. However, this huge rise in the quantities of generated data that needs to be processed, stored, and timely delivered, has created several new challenges.

In an effort to attack these challenges [Dom13] proposed a middleware with the concept of a Session capable of dynamically aggregating, processing and disseminating large amounts of data to groups of clients, depending on their interests. However, this middleware is deployed on a commercial cloud with limited processing support in order to reduce its costs. Moreover, it does not explore the scalability and elasticity capabilities provided by the cloud infrastructure, which presents a problem even if the associated costs may not be a concern.

This thesis proposes to improve the middleware's performance and to add to it the capability of scaling when inside a cloud by requesting or dismissing additional instances. Additionally, this thesis also addresses the scalability and cost problems by exploring alternative deployment scenarios for the middleware, that consider free infrastructure providers and open-source cloud management providers.

To achieve this, an extensive evaluation of the middleware's architecture is performed using a profiling tool and several test applications. This information is then used to propose a set of solutions for the performance and scalability problems, and then a subset of these is implemented and tested again to evaluate the gained benefits.

Keywords: Session, cloud, Big Data, DDDAS, cloud management providers, JPA.

Resumo

Hoje em dia, o elevado ritmo de produção de dados criou uma oportunidade sem precedentes para a extracção de informação em muitos domínios. Este aumento massivo nas quantidades de dados heterogenias que precisam de ser processadas, guardadas e entregues atempadamente, criou uma panóplia de desafios que precisam de ser resolvidos, tal como a gestão eficiente destes dados e a sua análise.

Num esforço para atacar este problema [Dom13] propôs um *middleware* com o conceito de Sessão, capaz de agregar, processar e disseminar grandes quantidades de dados dinamicamente para um conjunto de clientes, dependendo dos seus interesses. No entanto, este *middleware* está lançado numa *Cloud* comercial com capacidade de processamento reduzida de forma a reduzir os custos. Para além disso, este também não explora os conceitos de escalabilidade e elasticidade oferecidos pela infraestrutura da *Cloud*, o que gera um problema, mesmo que os custos não sejam uma preocupação.

Esta tese propõe-se então a melhorar o desempenho do *middleware* e a adicionar-lhe também a capacidade de escalar quando dentro de uma *Cloud*, requesitando ou dispensando instancias. Adicionalmente, esta tese também aborda os problemas de escalabilidade e custo explorando provedores de infraestrutura gratuitos assim como provedores de software para clouds *open-source*.

De modo a alcançar estes objectivos, é feita uma avaliação extensiva da arquitectura do *middleware* usando ferramentas de avaliação de software e várias aplicações de teste. Esta informação é depois usada para propor um conjunto de soluções para os problemas de desempenho e escalabilidade, das quais algumas serão então escolhidas para serem implementadas e testadas novamente, de modo avaliar os benefícios ganhos.

Palavras-chave: Sessão, Cloud, Big Data, DDDAS, provedores de cloud, JPA.

Contents

List of Figures

List of Tables

Listings

Acronyms

AWS Amazon Web Services. 15, 34

CEP Complex Event Processing. 2, 10, 21, 27, 30

CMP Cloud Management Provider. 6, 7

DDDAS Dynamic Data Driven Applications Systems. 1–4, 7, 10, 25–27

DSL Domain Specific Language. 28

EM Elasticity Manager. 21

EPL Esper Processing Language. 21

ESB Enterprise Service Buses. 28

FTP File Transfer Protocol. 29

IaaS Infrastructure as a Service. 6, 15

JDBC Java Database Connectivity. 29

JMS Java Message Service. 29

JPA Java Persistence API. 7, 29

JVM Java Virtual Machine. 33

LM Local Manager. 21

NIST National Institute of Standards and Technology. 11

PaaS Platform as a Service. 15

PCS Parallel Computation Systems. 22–24

PDS Parallel Database Systems. 22, 24

QoS Quality Of Service. 16, 17

RDS Relational Database Systems. 22

RFID Radio-Frequency IDentification. 2

RM Resource Manager. 21

SaaS Software as a Service. 15, 16

SDK Software Development Kit. 6

SEDA Staged Event-Driven Architecture. 29

VM Virtual Machine. 6, 14, 16, 22, 29

VoiP Voice over IP. 15

VPN Virtual Private Network. 12

WSN Wireless Sensor Networks. 2, 3

1

Introduction

1.1 Motivation

The current daily production of digital data is creating a non-precedent opportunity on information/insight extraction for many domains [MCBBDRB11], with data generators ranging from scientific and social media applications, large scale sensor networks, or business applications tracking consumers data [Jon12]. However, such massive data production is also raising many challenges on how its management can be effectively done. This spans the dimensions of data acquisition/access and persistence, data composition/filtering and processing, and data delivery/dissemination (the Big Data problem is detailed in 2.2).

For instance, solutions are required on when to access data from a large number of sensors deployed in a wide area (e.g. to save the limited sensors' battery) and where to save a continuous flow of data being produced by them. There is also the need to allow a simple and flexible way of combining different types of data and solutions for real-time data filtering/analysis (e.g. of data streams generated by sensor-like sources). Likewise, the dimension of static large-scale data processing is traditionally a challenge, as it is such data's access and delivery since it is highly costly, or even unfeasible, to move huge amounts of data to where it is needed.

One particular area where applications and systems have to address the problem of dealing with large amounts of heterogeneous data is Dynamic Data Driven Applications Systems (DDDAS) (see Section 2.3). DDDAS include applications such as Wildfire or Tsunami Forecasting, Flash flooding simulation and management, Weather Forecasting and Global Warming impact, Threat Management in Urban Water Supplies, etc. These types of applications require the inclusion/incorporation of dynamic data (either

live/fresh data or pre-processed data) from heterogeneous external sources and also from the application itself (self-feeding data). DDDAS demand thus not only big data storage and high-processing capabilities, but also that the infrastructure may be dynamically adaptable to respond to dynamic application requests and to a surge of sensing data.

Considering such adaptability challenges, Cloud Computing [MG11b] is establishing itself as a major player for supporting the above types of applications due to its characteristics on (a) elasticity on storage and processing power; (b) ubiquitous access (e.g. fundamental to mobile clients); (c) service-based resources' access (e.g. based on established standards supporting systems' integration); and (d) a pay-as-you-go model.

1.1.1 Open Challenges in Big Data/DDDAS Applications

The Big Data/DDDAS applications in its different dimensions, and their current Cloud-based solutions in particular, still present several open challenges, which have been the research subject of several recent works. We highlight next some examples concerning data access and aggregation, real-time data processing and analysis, and data dissemination.

First, the increasing diversity on data sources like Wireless Sensor Networks (WSN), Radio-Frequency IDentification (RFID) [CKR07] or mobile devices, require strategies that may hide such heterogeneity. Solutions based on the service paradigm aim to provide a simple and uniform access to such heterogeneous data sources [KBMBCDGFKMSSS02; Res07] and the Cloud context is a natural deployment environment for such service-based applications due to its service model conformance.

Second, one popular solution for data aggregation plus dissemination is the *Mashups* concept that supports composing information from multiple heterogeneous services and to provide its access as a combined-service [MCBFR08; AGM08]. This allows a simple and flexible way to combine different types of data/services, but mashups also require support on data integrity, security, and on sharing and reusability [KTP09; SM12].

Third, the real-time data filtering/mining, e.g. of data streams generated by sensor-like sources, has been tackled in the domain of Complex Event Processing (CEP) [WDR06; MC11]. Whereas traditional database management systems only support queries on indexed persistent data and only serve user requests on-demand, CEP also targets the problem of mining non-structured data and processing continuously a flow of incoming data. Examples of challenges in this context are the sequential access to data within time limits (as opposed to file's random access), and the consequent need for large memory and processing power support [MC11]. Several solutions for CEP in Cloud environments have been recently proposed as way to support the dynamic provision of resources necessary to online data streaming processing [AKS12].

Finally, several applications and systems in these domains increasingly require support on context building and sharing to a set of users with common interests. For example, big data generators and consumers like social networks and crowd sourcing applications build on content sharing among a set of users with common interests. Likewise, in emergency

prevention and damage contention applications, all teams involved may benefit from having a common view on what is happening in an endangered zone [BGP12; Dom13].

Additionally, the above types of applications still require novel ways on adaptability support based on their context's evolution [BGP12].

Context for these kind of applications may include, for instance, the status of

- data sources (e.g. battery's status of WSN);
- sensing data (e.g. the precipitation values at some point in time above 50 mm indicating a high probability of rain);
- computational resources (e.g. high overhead on data processing or accumulated cost/debt in the Cloud provider);
- communication medium (e.g. only a low signal WiFI is available for application's clients);
- existing clients (e.g. the autonomy of mobile clients or their geographic location, e.g. if they are accessing an application from a nearby area).

In turn, related examples of required adaptation/dynamic capabilities are

(a) dynamic flexibility on data sources' access (e.g. in Winter time automatically reduce the frequency access to a temperature WSN deployed in a forest in order to preserve the sensors' autonomy, but automatically increase that access if a high probability of fire is detected);
(b) real-time selection of (needed) data sources (e.g. access to WSNs for wind speed and direction, besides temperature, when a fire is imminent);
(c) data aggregation and filtering (e.g. modify the filtering functions in order to include the processing of novel/additional types of data);
(d) flexibility on data processing (e.g. elastic processing power only as needed);
(e) adaptive dissemination (e.g. reduce the amount of data sent to a mobile client with a reduced autonomy in order to preserve its battery).

1.1.2 A Session-based Abstraction for Delimited Data Management

Considering the aforementioned requirements on dynamic data management, especially in the context of DDDAS applications, there was therefore a need for adaptable solutions that might provide

1. flexibility and reusability on data access, data processing, and on processed data;
2. lower cost data sharing and context building;
3. more efficient ways of data dissemination, e.g. depending on users' context (e.g. data multicast to a set of users located in the same area or having the same interests).

The DDDAS applications, in particular, also require support for a feed-back loop on data management. For instance, live and pre-processed data are incorporated into simulations and these may dynamically select the relevant data sources that will in turn influence the simulation's execution.

Moreover, DDDAS applications require support for "people in the loop", e.g. in the sense that people are also data sources (e.g. via sensors on their mobile devices) and that asynchronous events generated by people influence the applications' execution.

To this extent, the work by [Dom13] describes a middleware deployed on the Cloud that responded to some of those problems, and which is the basis for the work proposed in this thesis (see chapter 3.1 for a detailed description of the middleware). The middleware supports the reduction of the processing, storage, and dissemination requirements on accessing a set of data sources, by allowing the saving and reuse of that processed data, and its dissemination to several clients at the same time. Such is also enhanced via the middleware's support on dynamic adaptation on that data management context changes.

Specifically, the middleware implements a Session abstraction for delimited data management in the sense that it builds a common context for a set of related clients, i.e. with common interests in terms of which data sources should be accessed (and when), which on-line data filtering is relevant, and how that processed data is to be disseminated.

A particular session, at some point in time, gathers data from a set of heterogeneous data sources, process this data according to some user-defined rules, save this data in a repository and send it to all existing clients in the session at that moment. In this way, individual users do not have to interrogate the data sources independently, but the accessed and processed data is reused to all clients. Data in the repository is available to clients that join the session at a later time, reducing in this way the need to re-process the same data and allowing these later clients to build the same context view as the other clients already in the session.

Clients may also behave as data sources by uploading data into the session.

Additionally, a session provides its clients with dynamic and automatic support on

a) data sources' access and selection via rules;
b) changing the processing rules as needed;
c) data dissemination (e.g. a set of rules define how data is disseminated to clients, for instance via a stream of data or a via a publish-subscribed model).

Novel rules may be added during the session's execution time and the execution of some rules may be automatically triggered based on data values from the data sources. A replay of the whole session's actions is also possible since all data events within the session are preserved in the repository, e.g. allowing a post-mortem analysis of events when a session is being used to coordinate a set of users in a emergency situation like a flash flooding event in urban environments [Dom13].

1.2 Problem

All the characteristics of a session point hence to the need of a high processing power support when the number of data sources is large or data is generated at a fast rate. Likewise, a large number of clients require computing and dissemination capabilities able to deliver information in useful time, and to respond to a unexpected peak number of clients. The session's repository also has to be scalable to cope with a surge on data generation and to respond to worldwide client's accesses.

Being deployed in the Amazon Cloud, the middleware described in [Dom13] should benefit from the Cloud's capabilities in terms of scalable computational and repository resources (section 2.1 describes Cloud computing characteristics).

However, a first evaluation of the middleware's architecture, as described in section 3.2, and a first small and non-systematic evaluation of its performance response, led us to the hypothesis that the middleware would present serious limitations on the following situations:

a) data sources producing data at a very high rate;

b) a large number of data sources per session;

c) a large number of clients per session;

d) a large number of active sessions.

This hypothesis was the basis for the work described in this thesis and represents the following sub-problems:

1. identification of the major contention points of the middleware, and which also prevent exploring the scalability features offered by Cloud platforms;

2. how to improve the middleware's performance if running on a single (real or virtual) machine;

3. how to adapt the middleware running in a Cloud context in order to effectively benefit from its scalability features;

4. how to run the middleware in a free Cloud so that the middleware may be executed in the future in available academic contexts in order to a) reduce possible costs always present in commercial clouds and b) so that the middleware may be used to process data already accessible from other platforms and hence avoid replicating data to the Amazon (e.g. situations when it is unfeasible to move huge amounts of data).

In the next section we describe our proposed solution that responds to the problems cited above.

1.3 Proposed Solution

The proposed solutions for the identified four sub-problems are (following the same order):

First, to perform an extensive evaluation of the middleware and analysis of each one of its components. This evaluation can be done using profiling tools, and once completed, will serve as a foothold in solving all the other subproblems as it will provide all the necessary data to take the right decisions.

Second, to improve the middleware's efficiency while still inside the same machine by using additional threads to explore parallel strategies capable of speeding up the execution times of the middleware under certain rules. This can range from adding additional Java threads, to replacing certain components of the middleware with parallel equivalents. This may allow the middleware to make a more efficient use of its resources and consequently decrease the number of needed instances if deployed in a cloud. It is important to stress however, that if the machine where the middleware is deployed only has a single core, using a multi-threaded solution will not affect the performance of the middleware positively, in fact it may even harm it since the single core now has to divide its attention by multiple threads. In the specific case of this thesis however, we only used architectures with multiple cores, and so this proposed solution is viable.

Third, to add a communication layer that allows the middleware to use the Java Amazon Software Development Kit (SDK) in order to give it control over the creation and deletion of additional Virtual Machine (VM)s. In addition to that layer, a monitorization layer is also needed, in order for each active instance to be able to check its current load levels and with that make a decision. Such decisions may be done using different strategies, such as the master and slave strategy or the peer to peer strategy.

Fourth, to use a free Infrastructure as a Service (IaaS) structure together with a free Cloud Management Provider (CMP) (see Section 2.1). Such deployment will eliminate the costs of using commercial clouds, while also adding the additional benefit of more control over the data.

1.4 Contributions

This thesis's contributions are focused on the improvement of the middleware's performance inside a single machine, and on its deployment in a cloud in order to achieve both elasticity and scalability. The work will be guided by the following points:

- Analysis of the middleware's main components and their adaptation in order to support parallel execution. The contribution is the implementation of parallel components tackling the current major problems of scalability and elasticity present in the middleware.
- Deployment of a modified version of the middleware in the Amazon cloud, with performance improvements and scalability capabilities.

- Analysis of a CMP allowing an easy migration from the service deployed in Amazon to an open-source solution.
- Analysis of a Java Persistence API (JPA) implementation capable of supporting NoSQL for integration with a CMP.
- Writing of the dissertation document discussing the selected implementation.

1.5 Document Organization

Chapter 2 briefly explains the main areas involved in this preparation. Thus, it starts by presenting the cloud technology, followed by a brief introduction to the Big Data and DDDAS areas which bring challenges that the middleware addresses. We end this chapter with a section about Apache Camel, the integration framework used in the middleware, and another one about metrics, which describes some concepts and tools used when evaluating its performance.

Chapter 3, describes the main characteristics of the middleware and the test platform chosen to evaluate it. This chapter also presents the results of the conducted tests together with their performance and operational analysis, in order to justify the performance improvements.

This is followed by Chapter 4, which proposes solutions to the problems identified in the previous chapter, and continued by Chapter 5, which details the implemented solution as well as other dimensions that have to be taken into account concerning the middleware's deployment and the several options concerning the available cloud providers.

We end this thesis with Chapter 6 by wrapping up the work done and suggesting future improvements to the middleware.

2

State of the art

With a virtually unlimited amount of outwards scalability and with its high flexible nature, the Cloud has become a major player in any system desiring to go global, or to encompass a large number of clients possibly over multiple regions without fixed restrains of machine numbers and specifications. In order to clarify its main characteristics and its high relevance as a deployment platform, Section 2.1.1 presents a cloud definition, Section 2.1.2 follows with the characteristics of the most common cloud types, Section 2.1.3 presents the underlying architecture shared by all those cloud types, and Section 2.1.4 builds on the previous information to present the most common cloud business models. We finally end this introduction to the cloud with Section 2.1.5, which details the main characteristics of a cloud.

With this in mind, we discuss Big Data in Section 2.2. Nowadays, the amount of data created by systems and their interactions is too big and diverse for common systems to use. Big Data comes in high volume, has elevated throughput and is usually very diverse. In fact, the middleware was designed to accommodate Big Data, by allowing the dissemination of diverse data to multiple clients in a fast and efficient way. However, such does not yet happen due to implementation problems that lead to efficiency drawbacks - and that is the main problem this thesis addresses. Thus, this need to effectively address Big Data is one of the most prominent factors in bringing the Cloud into play. With its previously mentioned scalability and flexibility attributes, the Cloud has become a standard solution to systems that must process Big Data in an irregular fashion that is hard to predict. Thus, in order to introduce the reader to the challenges of Big data, we start with Section 2.2.1, which describes the main aggregation strategies used to process and organize Big Data, and we end with Section 2.2.2, which details the approaches used to face the problems brought by Big Data using the Cloud.

One of the areas where the processment of Big Data is required, is the DDDAS area, briefly presented in Section 2.3. These applications require the management of large amounts of data in real time. The middleware attacks challenges from this area by employing a dynamic solution that allows clients to access data from different points in time, originated from different data-source types, in a convenient manner. Therefore the DDDAS applications are a good use case for the middleware. Section 2.3.1 introduces the reader to the main benefits and challenges in this area, and Section 2.3.2 concludes how applications from this area can benefit from being deployed in the cloud.

Then we move to the technologies used in the making of this thesis. This encompasses our last two Sections, an introduction to Apache Camel (Section 2.4) and an introduction to metrics and profiling tools (Section 2.5).

Apache Camel is the integration framework that was used in order to implement the specifications of the middleware. This framework has a rich ecosystem of components that were used, specially the Esper component, which allows the middleware to have CEP. Thus, Section 2.4.1 explains what exactly is Apache Camel, Section 2.4.2 brings forth the main benefits of using it and Section 2.4.3 presents Camel's main components. We end with Section 2.4.4, which clarifies why and how this framework is used in the middleware.

When complex systems use many different technologies at the same time, efficiency issues may arise. To deal with such issues one must first identify them. This last Section covers some basic notions of software metrics (Section 2.5.1), as well as the profiling tools used to find these issues in the middleware when deployed locally (Section 2.5.2) and in the cloud (Section 2.5.3).

2.1 Cloud computing

2.1.1 Cloud definition

Nowadays, more than anything else, Cloud is a buzzword that can have many meanings. According to the authors of [FZRL08], the Cloud is mainly a business model, in which the providers offer resources that can be either physical or logical and that has a policy of on-demand usage coupled with a and pay-as-you-go model for the client. Furthermore, [VRMCL08] additionally defends that the services offered by these providers should have a high level of scalability, while others defend that a cloud has to be a set of virtualized resources to fulfill these demands.

Some even go further as to consider the cloud as a family member to the grid computing system, which causes some attrition with the specialists from the area [VRMCL08], that even though agree that there are some similarities between the two, still defend that they two different systems.

Therefore, given the fact that there are many definitions of cloud computing, and that there is no overall consensus inside the community as to what the cloud really is, we have decided to build our own definition of the cloud, based on the previous opinions

and on the definition provided by National Institute of Standards and Technology (NIST) [ZCB10]:

"*Cloud computing is a business model, in which the providers offer resources that can be either physical or logical and virtualized, with a high level of scalability and that are automatically charged with the pay-as-you-go policy, thus following the general premises of utility-computing.*"

2.1.2 Cloud types

Cloud computing has many types, the main three being public clouds, private clouds and hybrid clouds. However there are also types of clouds, like the private virtualized cloud, community cloud and even the intercloud recently announced by Cisco Cloud Computing [Cis].

In this section we present a basic definition, together with the main advantages and disadvantages from each of the main types of cloud. It is however, important to notice that because the cloud is a new concept, and is yet to became mature, many of these definitions are likely to evolve with time due to a better understanding of the cloud technology as it ages.

2.1.2.1 Public cloud

Focuses on sharing or renting services and infrastructures from external entities with several clients. In this type of cloud the resources are usually allocated dynamically during an undetermined amount of time.

Advantages

- The client does not need initialization money to build the cloud because this money has already been invested by the supplier;
- Migrates the responsibility and management of risks to the supplier;
- Are usually a lot bigger than the private clouds;
- The client can have access to state of the art technology without having to directly invest in it.

Disadvantages

- The client has no control over the data, security and network.

2.1.2.2 Private cloud

Also known as internal clouds, these are designed for the exclusive use of a company. Depending on the contract, it may be maintained by the company itself, or by an external company. These clouds usually exist when the client has specific needs or when it wants a long term contract.

Advantages

- The client has more control over the infrastructures, and can therefore have its own security and failure protocols;
- Easier and cheaper than buying a database.

Disadvantages

- Criticized by being similar to private server farms, thus sharing all their problems including maintenance;
- Requires an initial investment to buy the necessary infrastructures.

2.1.2.3 Hybrid cloud

This is a combination of the public cloud with the private cloud, trying to overcome the limitations of both types while gaining the benefits from them. As the name suggest, in this type of cloud, part of the system is external to the client, while the other part is internally managed by him.

Advantages

- More flexible than a private cloud or a public cloud;
- Provide more control than public clouds have easier scalability when compared to private clouds;
- Helps isolate and delegate the computational resources used, thus increasing the availability of the company's own resources.

Disadvantages

- Requires a careful analysis of what should go to the private cloud and what should go to the public one;
- Moving big amounts of data between the two clouds results in additional costs and stateful applications are harder to maintain.

2.1.2.4 Virtual private cloud

This is an alternative to attack the problems present in both private and public clouds. In this approach, the supplier of a public cloud has a has system of Virtual Private Network (VPN)s that allow him to define management and security protocols, thus effectively simulate a network of virtual servers and their connections. This is sometimes mistaken as a hybrid cloud, however the main difference is that in such a cloud, both public and private clouds are real, while in the one the private cloud is merely virtualized on the top of the public cloud.

Advantages

- Allows the client to use the provider's infrastructures as part of his own cloud;
- Allows the dynamic manipulation of the protocols defined;
- Easier to scale because the created servers are virtual.

Disadvantages

- Adds the complexity of migrating workload from the client to the supplier's cloud.

2.1.2.5 Community cloud

This type of cloud is a shared cloud between several entities that share a common interest, such as politics, security or other.

Advantages

- Because it shares the costs, this option is the cheapest one.

Disadvantages

- Because each company has specific security restrictions, this type of cloud has to manage them all at he same time, consequently not being very effective at doing it;
- There is little control over the resources because the environment in which they are is shared.

2.1.2.6 Cisco Intercloud

This is a new type of cloud being developed by Cisco. According to them, the intercloud will emerge as a public solution, open and uncoupled, like the Internet.

Cisco goes even further and defends that this intercloud can even be seen as an extension to the Internet because it will separate content from client, like the Internet currently does (i.e, there does not need to be a previous contract to access content), and it will also separate consumers from suppliers.

Together with this, Cisco also predicts that workload migrations will be one of the main challenges in creating this intercloud.

2.1.3 Cloud architecture

The architecture of a typical cloud can be divided into four distinct layers:

- Hardware layer;
- Virtualization layer;
- Platform layer;
- Application layer.

2.1.3.1 Hardware layer

The hardware layer is responsible for the management of the physical resources of the cloud. This means the servers, routers, switchers, cooling and power systems. This is usually in one or more data-centers, which contain several servers deployed and connected in racks.

This layer is also the one responsible for the hardware configuration, fault tolerance and network bandwidth management. Consequently, this layers represents all the physical structure necessary for building a public cloud, and thus represents the majority of the initial investment. By using a public cloud the client does not have to worry about any of this, but it eventually ends up paying part of these costs with the rent fees.

2.1.3.2 Virtualization layer

Also known as the infrastructure layer, this layer is responsible for creating a pool of resources, which is achieved by using tools such as KVM, Hyper-V and VMWare. This is essential to allow the dynamic allocation of resources. Furthermore, because these tools create VMs, the user then has the freedom to install any software platform he likes on the virtual machines, thus giving him more flexibility.

A good example of a provider operating in this level is the Amazon Web Services cloud, which allows the user to create a VM, and then customize it to his needs.

2.1.3.3 Platform layer

The platform layer is built on top of the virtualization layer, and it consists in operative systems and frameworks for the most diverse kind of applications. This is aimed at minimizing the work that is needed to deploy, launch and effectively use a create machine, and it varies from provider to provider.

Examples of a providers operating in this level are the Google App Engine, Windows Azure and Heroku. Google App Engine offers APIs, such as the python and Java APIs, and frameworks, such as the webapp2 framework [Web], that allow the users to build web applications and websites by using the system and the predefined configurations set by them. Windows Azure is similar by providing runtime services for .NET based applications, and Heroku allows the management and deployment of ruby applications with the ruby on rails framework.

2.1.3.4 Application layer

Finally, in the top of the architecture, there is the application layer, responsible for the applications themselves, such as Youtube, Facebook, Google Docs and others. These specific types of applications can take advantage of the clouds unique scaling and flexibility characteristics, thus automatically adapting themselves to the current situation. Consequently, this allows for better performance, more availability, and lower maintenance costs.

2.1.4 Cloud business models

Now that the architecture of the cloud is known, it is time to present the business models that were created for the specific layers of this architecture.

2.1.4.1 Infrastructure/Hardware as a Service (IaaS, HaaS)

This business model represents the general idea of renting infrastructure to customers. This infrastructure can be either real, or virtualized, and it allows the clients to pay only the necessary as they grow with time.

However, this business models presents some unique challenges, such as:

- The client has to know how the software being installed will react when inside a dynamic environment where the hardware scales;
- The client must figure a way for the software to take maximum advantage of the scalability presented;
- The client must understand how the work load inside the cloud is managed and draw the application around those limitations;
- The client must manage energy and bandwidth costs;
- The client must design fail safe strategies for the software.

Examples of IaaS providers are Amazon Elastic Compute Cloud (part of Amazon Web Services (AWS)) and GoGrid.

2.1.4.2 Platform as a Service (PaaS)

While the previous model is built on top of the Hardware and Virtualization layers of the cloud architecture, this layer is built on top of the Platform layer. Here, the main idea is to give programmers a software development platform that includes all the programs and systems they need to create software, from the development stage, to the implementation and testing of a project.

In many cases an IaaS provider will also offer a Platform as a Service (PaaS) provider, so both them are confused at times. However, a PaaS provider is different in that it has to:

- Offer one or more programming languages. These languages have to be used by the programmers and usually come with integration APIs that allow the programmers to have more control over the scalability of the application;
- Offer a system or a dashboard to allow the programmers to check the usage of the system;
- Separate each client's development environment. This is usually done by resorting to virtualization;
- Implement communication protocols such as XML, Json or SOAP, or have a public API;
- Have well defined and documented interfaces, for users have the ability to build on top of them.

2.1.4.3 Software/Application as a Service (SaaS, AaaS)

Lastly, software/application as a service refers to the model where the provider allows user to use certain applications on demand. Because these applications are built on top of the other layers, they also scale depending on the amount of users currently using their services. This includes applications such as Voice over IP (VoiP) and other referred cases such as Youtube and Facebook. Some of the main characteristics of Software as a Service (SaaS) are:

- Integration requisites with other applications;

- They have to be modular and service oriented in order to allow other services and companies to use them;
- Composition of the different types of technology used, such as J2EE, .NET, Hibernate, Spring, scalable infrastructure and other services;
- They usually evaluate scalability, fail tolerance, security, user shared environments and ease of configuration before being deployed to the production environment;
- SaaS applications usually are generalist in order to promote a large audience of users;
- They must have public and documented interfaces;
- They may have a billing system that allows user to pay for the system, or upgrade to a premium account;
- They must be capable of deploying software updates without compromising stability.

2.1.5 Cloud characteristics

Cloud computing and management is different from the traditional management of individual datacenters in the following ways:

Multiple allocation - a cloud datacenter hosts services from multiple clients. Cloud architecture provides for a natural division of responsibilities into specific layers, i.e., the owner of the datacenter only focuses on the infrastructure layer while the owner of the service only focuses on the service being deployed (service layer). However, such division also creates interaction problems between the layers of responsibilities, mainly because each layer is completely unaware of what is happening in the others.

Aggregation of shared resources - the owner of the infrastructure can aggregate multiple services into only one physical machine, by deploying each service in a different VM. This minimizes cooling and energetic costs, consequently increasing the number of services that can be supported.

Geographical distribution and ubiquitous access - as long as a user has an Internet connection he will always be able to access a cloud service, no matter where he is. To support this feature clouds usually have several datacenters deployed all over the world. However, even though sometimes users can access the cloud servers, the quality of service can be poor, and there are also legal considerations that must be taken into account depending on the country.

Service orientation - each cloud provider presents the services with contracts detailing the minimum Quality Of Service (QoS), legal obligations, and other aspects, thus making the customer's satisfaction a critical objective.

Resource elasticity - because services run on VMs, the provider is then free to allocate more VMs to a certain service is such is needed, or to dismiss VMs if the service is idle. This allows a dynamic adaptation directly linked to the level of work required. In traditional datacenters, such is not possible - the number of machines adjudicated to a given service is static.

Price - cloud providers apply a *pay-as-you-go* policy, in which the customers only pays for what they use. Consequently the costs scale with the client's business and according to the customer's needs.

Consequently, thanks to these characteristics cloud deployment brings the following benefits to the customers:

Flexibility - the ability to scale up to face stress conditions, or to scale down to reduce costs, allows the application deployed to be both cost effective and still maintain a high quality of service.

Energy friendly - as a direct consequence of the previous benefit, and because the customer only uses a minimal amount of resources to maintain its QoS, this has a positive impact in energy savings and therefore in the carbon footprint, which can go up to 90% [sal12].

Delegate responsibilities - because the middleware infrastructure can be delegated to a third-party, the developers ca invest less time in it. Furthermore, some of the companies that host the infrastructure also have disaster recovery plans, periodic backups and other protocols which can all be discussed with the contracts.

Ubiquitous access - as previously mentioned, one can access the middleware in any location. Some cloud providers, like Amazon, even allow direct control over the deployment of the infrastructure for faster access times and lower fees.

Thanks to these attributes, cloud computing has established itself as an infrastructure solution for tacking applications that deal with Big Data. Therefore, the next chapter presents the nature of Big Data and how the Cloud can be seen as solution for the problems brought by this area.

2.2 Big Data

Today the amount of data created within the digital world is far too big and diverse for common systems to handle. According to [Mew12], this kind of data - Big Data - cannot be analyzed in a traditional database because a traditional database system does not have the capacity to handle large-scale data. Furthermore, the previously mentioned article defines three main characteristics for big data:

High Volume - big data's volume is too much for common data centers, slowing them down and interfering with their quality of service;

High Velocity - big data is often streamed (videos or music) or can be time-sensitive;

High Variety - big data tends to be a mix of several data types, which are unstructured, uncorrelated, and can be produced by many heterogeneous data sources.

According to the same study, big data's complexity is handled in four dimensions: (a) Deep Analytics; (b) High Scalability; (c) High Flexibility; (d) Real-time.

Because each of the previously mentioned pillars would alone be enough for a thesis on big data, the following sections will only focus on the already existing approaches that

have been taken to handle high scalability and flexibility. This is important because the middleware will receive big amounts of data from external sources, and must therefore be able to process it quickly in order to disseminate it to the clients. Therefore, the following research shows light on the solutions that already exist, and what can be learned from them.

2.2.1 Data Aggregation and Filtering Solutions

2.2.1.1 Mashups

Nowadays with the explosion of content and services in the Internet there is more digital content, such as photos, videos, audio, etc, than has ever been before. This content is usually provided by services that are dedicated to acquiring it and publishing it. Mashups, also known as composite services, are applications that can combine the output of base services and aggregate it in a more presentable fashion[SM10; BT11], allowing the user to filter, combine and modify data retrieved from multiple sources [SM12]. Because mashups allow the user to choose the necessary sources, they promote the development of lightweight applications with lower development costs associated, thus forging a path of greater harmony between business and IT [BT11].

Mashup classification

Because mashups are a recent technology being investigated there is still no consensus on their types and classification. Nevertheless, by combining the ideas of [SM12; SM10; BT11; SFDM12] the following classification types can be derived:

Data Acquisition - evaluates the way in which the mashup service acquires information from its base services:

- **Data Mashup** - interacts with base services through the classical Call-Response paradigm;
- **Event Driven Mashup** - interacts with base services through the Event-Notification paradigm.

Processing Location - evaluates where the majority or the most important sections of the processing work are done:

- **Client-Side Mashup** - the main processing runs on clients like web browsers and smartphones;
- **Server-Side Mashup** - the main processing runs on servers, which can be physical machines (such as servers in a cluster) or virtual machines (like instances controlled by VMWare vSphere).

API Number - classifies mashups based on the number of APIs they use:

- **Multiple API Mashups** - use and combine multiple APIs from multiple base services;
- **Single API Mashups** - use only one API, and provide a better visualization model for the base service.

Aggregation Visibility - consider socio-economic features of a mashup ecosystem, such as its economic purpose, set of users, API relevance, etc:

Explicit Mashups - focus mainly on ways to get profit for the mashup provider. Can be divided into two groups:

Commercial Mashups - creates profit by selling the mashup service to the users in the open public;

Enterprise Mashups - creates profit by solving a specific business-related problem for a company.

Implicit Mashups - focus on the reasons that led to the creation of the mashup or evaluate the main focus of the mashup.Can also be divided into two groups:

Situational Mashups - appear when a small mutually trusted set of users created a mashup for a specific purpose for a small amount of time;

Essential API Mashups - classifies the mashup by analyzing which API is more important in its functionality and/or architecture.

Event Driven Mashup architecture

Because of their data driven nature, these Mashups are the ones of most interest to this thesis's project. Therefore, in this section we proceed with a quick analysis of their life cycle and then we evaluate how its architecture supports it. According to [SM10], this life cycle is divided into the following stages: 1. Mashup Creation; 2. Mashup Deployment; 3. Mashup Activation / Execution; and 4. Mashup Management.

With the life cycle studied, we can now evaluate the generic architectural components [SM10; SFDM12; BT11] that support them:

User Interface - manages the interaction with the user allowing an easier Mashup management;

Service Creation Environment - supports mashups's creation by developers;

Mashup Container - manages mashup deployment and execution. It is divided into two main components:

Service Execution Platform (SEP) - responsible for managing the mashup execution. It subdivided into:

Service Proxy (SP) - converts the base service's information into something the Orchestrator can understand, by converting between protocols and data formats;

Orchestrator - responsible for the execution and main processing. It is protocol agnostic thanks to the SP;

Communication Interface - the protocol that the SP and the Orchestrator use to communicate between each other. It has two main primitives, the **invokeAction** used by the Orchestrator to issue command to the SP at run time, and the **notifyEvent** used by the SP to communicate changes and state to the Orchestrator.

Deployer Module (DM) - responsible for managing the installation of mashups into SEP. It is subdivided into:

Mashup Deployer - which performs operations related to the mashup deployment phase;

Service Deployer - which performs operations related to the service deployment phase.

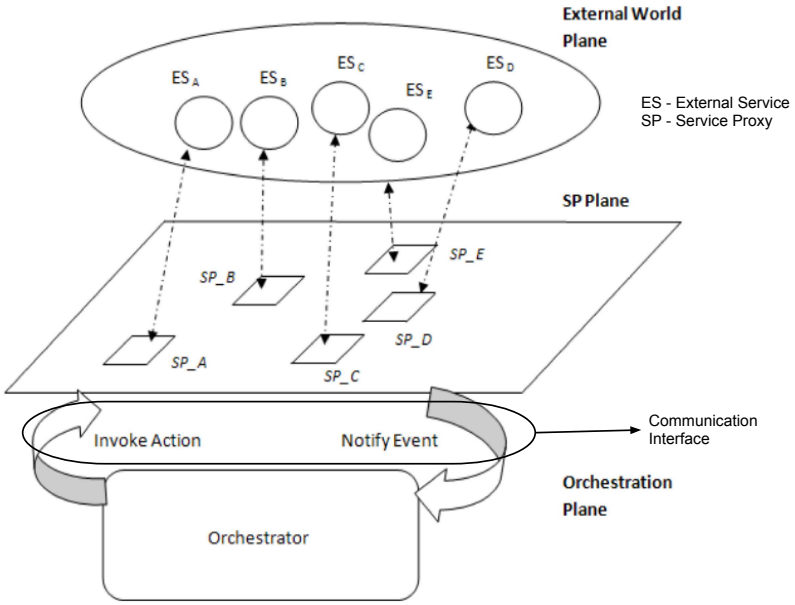

Figure 2.1: Architecture of the Service Execution Platform [SM10]

Mashup challenges

Because Mashups are a new area under great development, it has some challenges that need to be addressed carefully, like [SM12; SFDM12]:

- Mashup security, considering both the protection of sensitive data and management of user roles;
- Legal problems, since some services do not allow their data to be used but web scrapping or HTML crawling can still retrieve it;
- Constant changes in the field of web programming;
- Mashup development tools are underdeveloped and the major contributions made by Google and Microsoft have been discontinued.

Still, the power that mashups bring is decisive for many users and applications and efforts are being done in the academic level to tackle those challenges at the most various levels [SFDM12].

Mashups are important for the middleware in the sense that the middleware too aggregates information from several different data-sources in the context of a session 3.1.1. However, the middleware does not address mashups specific challenges nor is it intended to. For all purposes, no matter how much can be learned from mashups, the middleware has a different purpose.

2.2.1.2 Complex Event Processing

Additionally, [PRC13] refers CEP as a solution to Big Data. Consequently, CEP also becomes a solution that can be used by Mashups as well, specially by those that are data driven. Because the middleware deals with some of Big Data's challenges, it adopts this solution by using Esper queues in the context of the session. Esper is a component for CEP and event series analysis. Esper enables the rapid development of applications that process large volumes of incoming messages or events, regardless of their nature. It also filters and analyzes events using the Esper Processing Language (EPL) in order to respond to predetermined conditions [Esp].

The authors of [PRC13] go even further and define Event Clouds, and streams of events. Although these notions are not yet applicable to the middleware because the data it receives is not timestamped, this could be an improvement for the future. By having partially ordered and ordered data respectively, event cloud and stream events allow for the prediction of system states which improves the efficiency and prevents inconsistent states by detecting them ahead of time.

2.2.2 Cloud-based Approaches

2.2.2.1 Scalable Big Data Processing

Big Data approaches that deal with large amounts of data, scalability, and elasticity can be divided into two main groups:

Architectural - focuses on component's roles and their realization;

Technological - focuses on a paradigm and on a set of tools and algorithms that implement it.

2.2.2.2 Architectural approach

Originally defined in [GJPPMSV12] for the StreamCloud project, this approach defends an architecture for scalability and elasticity, which is heavily based on load balancing stragglers. It is divided into three main components:

1. Elasticity Manager (EM);
2. Resource Manager (RM);
3. Local Manager (LM).

Each worker runs a LM that monitors resource utilization and incoming load. Periodically, this information is reported to the EM, which then aggregates the information and decides if the system (or a subpart of the system) is in a deficit or a superavit of active worker. In the first case, the EM asks the RM for an worker from within its pool of inactive, on stand-by, worker. Because workers are not created from scratch this process is fairly quick and efficient. On the other hand, if the EM detects that there are too many workers running, it simply gives them back to the RM, which then decides if they should be put to sleep in the pool or if they should simply be disposed.

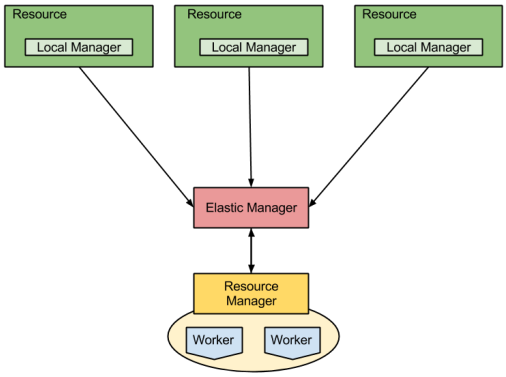

Figure 2.2: Architectural approach's components and interactions

It is important to note that although the work in [GJPPMSV12] does use this approach in a specific case, we can easily adapt it as long as each component behaves accordingly to one of the defined roles. In fact this approach is abstract enough to be implemented with VMs, threads, or any other concurrency abstraction, which is an advantage not shared by technical approach in Subsubsection 2.2.2.3. However, unlike the technical approach which allows programmers to use tools and algorithms for specific problems, this approach only provides guide lines for the programmer, leaving all the implementation's components, connections and concepts for him/her to implement and define.

It is important to notice that there are other distributed systems with similar balancing approaches, such as the Borealis project [AABCHLMRRTXZ05], upon which the work proposed in [GJPPMSV12] was built. However, although Borealis also has some scalability and distributed concerns, its main focus lies on database queries.

2.2.2.3 Technological approach

From the technological perspective there are four approaches to tackle Big Data problems that can be classified as follows [CCJOSVW12]:

1. Parallel Database Systems (PDS);
2. Parallel Computation Systems (PCS);
3. Map-Reduce;
4. Directed acyclic graph based systems (DAG-based systems).

Developed in the late 1980s, PDS was created to allow transparent parallel querying for databases deployed in static clusters assumed to never fail. However, today's applications run distributed systems comprising many nodes [CCJOSVW12; GJPPMSV12], which can fail at any time. Hence, there is an ever increasing need for scalability and elasticity. Consequently, PDS has been replaced in its majority by cloud and its scalable / distributed Relational Database Systems (RDS), such as Amazon RDS[1].

PCS combines the use of several tools (which can be frameworks, platforms and specific programming languages) for the purpose of creating high performance parallel

[1]Amazon RDS - http://aws.amazon.com/rds/

programs, usually for the scientific domain. An example of these tools are Open-MPI[2] for parallel computation on several nodes, and CUDA[3] to allow computation exploring GPUs. PCS has the advantages of being highly efficient and cost-effective if properly designed and implemented. However, on the downside it only provides low-level programming primitives and requires developers to manage all the problems associated with communication, load balancing, parallelization and node-failure on their own.

Map-Reduce [DG08] is a programming model that allows the computation of large and complex tasks in a parallel and distributed environment. A program in Map-Reduce has two phases:

Map - divides the task into smaller chunks that can be processed independently;

Reduce - combines all the results from the previously calculated independent tasks and forms the final output.

Complex programs usually combine several map and reduce stages until they reach the real final output. An open-source example of this model is the framework Hadoop[4]. Hadoop has the advantages [CCJOSVW12] of handling automatic division of job into tasks, automatic placement of computation near data, automatic load balancing, recovery from failures and stragglers and elastic scalability, thus allowing users to focus on logic and not on parallel computation.

However, the Map-Reduce model that Hadoop forces is relatively rigid and hard to adapt for some applications, which usually leads to a bad performance when compared to the other options if the program is forced into a model it was not designed to fit in the first place. This means that Map-Reduce is good for algorithms and programs designed for its specific two-phase model and little less. Furthermore it is also quite hard to learn and debug.

Finally, the DAG-based systems are frameworks that allow the user to specific a program's behavior through the design of acyclic graphs using a graphical user interface. Examples of such frameworks are both Dryad [IBYBF07] and Clustera [CCJOSVW12]. Both systems aim for the same objectives and according to [CCJOSVW12] differ only in implementation details.

A DAG-based model is general enough to implement other models such as Map-Reduce and relational algebra, handles job creation and management, resource management, job scheduling and monitoring, fault tolerance and re-execution just like Hadoop [CCJOSVW12], allows the user to change the graph dynamically on the run and it is also easier than Hadoop if we want to develop applications. However, due to the nature of the paradigm, its implementations have a complex and tedious interface [CCJOSVW12], thus leading to the side effect of having complex algorithm and computations represented in a way that most users wont understand.

Table 2.1 summarizes the advantages and disadvantages of each type of paradigm.

[2] http://www.open-mpi.org/

[3] http://www.nvidia.com/object/cuda_home_new.html

[4] http://hadoop.apache.org/

Paradigm	Pros	Cons
PDS	• Ability to parallelize queries in a transparent way to the user	• Strongly outdated and replaced by newer technologies
PCS	• Highly efficient and cost-effective	• Only provides low-level programming primitives • Developers must manage communication, load balancing, parallelization and node-failure
Map-Reduce	• Automatic division of job into tasks • Automatic placement of computation near data • Automatic load balancing • Recovery from failures and stragglers • Elastic scalability • Allows users to focus on logic and not on parallel computating	• Bad performance • Rigid and hard to adapt for some applications • Hard to learn and debbug
DAG-based	• General enough to implement other models such as Map-Reduce and relational algebra • Job creation and management • Resource management • Job scheduling and monitoring • Fault tolerance and re-execution • User can change the graph dynamically on the run • Easier to adapt to the needs of an application than Map-Reduce	• Complex and tedious interface

Table 2.1: The pros and cons of technology paradigms for Big Data

All these approaches used to tackle Big Data challenges have something that the middleware can learn from. However, given the dynamic nature of the middleware, not all can be used. For example, Map-Reduce does not produce real time results, and the DAG-based system simply is not applicable. Thus one is left with the PDS and PCS technical solutions. The PDS, now replaced by RDS, has scalability issues, addressed in 3.2.2, so PCS is the only one left, and the technical approach that more resembles the set of solutions proposed 4.1.1.

2.2.2.4 Locality-awareness

Big Data approaches and challenges related to this work do not end here though. Another important concept, explored in [XGS11], is co-location. Co-location is the concept of having two or more services within the same cluster or cloud provider taking maximum advantage of the infrastructure where they are deployed. For example, if a touristic photo service and a map service are both deployed in Amazon, then they should be deployed not only in the same physical datacenter, but should also be able to directly fetch information from Amazon's virtual machines using the datacenter's LAN connecting all the physical hosts.

Such approach would mean that both services would benefit greatly from the speed improvement provided by the datancenter's LAN and would not have to pay data transfers from one datacenter to another, thus providing a faster service with lower costs for the client.

A concept that can be used together with co-location is the concept of view as defined in [GGL09]. This concept allows different services to share a portion of their data publicly with other services and to keep the rest private or not shareable. In fact this concept derives from the database world, and it would greatly benefit from co-location.

Other concepts, such as geo-distribution are also important for the project. Because our application is meant to support various users, independently of where they are in the world, geo-distribution of data also becomes a concern. If the data that the users are trying to access is far away from them, then their quality of service will be diminished. To solve this problem [XGS11] mentions project Volley, which could be used to fix the problem of geo-distribution. However this would heavily depend on the cloud provider and in its distribution of resources around the world.

One of the areas of applications that require the processment of large quantities of data in real time are the DDDAS. Thus, the next Chapter introduces this applications, how they use such amounts of data, and how it all links back to the Cloud.

Both co-location and geo-distribution are important concepts for the middleware. Geo-distribution is important in the sense that we want the middleware to be accessible anywhere in the world. However, because the middleware is deployed in the Amazon cloud (see Chapter 5), geo-distribution and visibility are an issue, mainly because instances deployed in Europe are not visible to instances deployed in America. Although Amazon and other cloud providers do have ways of surpassing this difficulty, they usually involve transferring data from datacenter to datacenter, and that can be costly.

As for co-location, the middleware's session abstraction 3.1.1 already promotes this concept. Sessions aggregate data for clients sharing a set of interests. Hence, clients within the same session will be using middleware's resources from the same location, i.e. from the same datacenter.

2.3 DDDAS

DDDAS are defined in [Dar12b] as applications with the ability to dynamically incorporate additional data (either on-line or preprocessed) into their own execution and to select and tune the most adequate data sources, at some point in time. Thanks to this dynamic nature, DDDAS are heavily used in computational science applications and are specially useful in those that encompass several areas and need several experts from them.

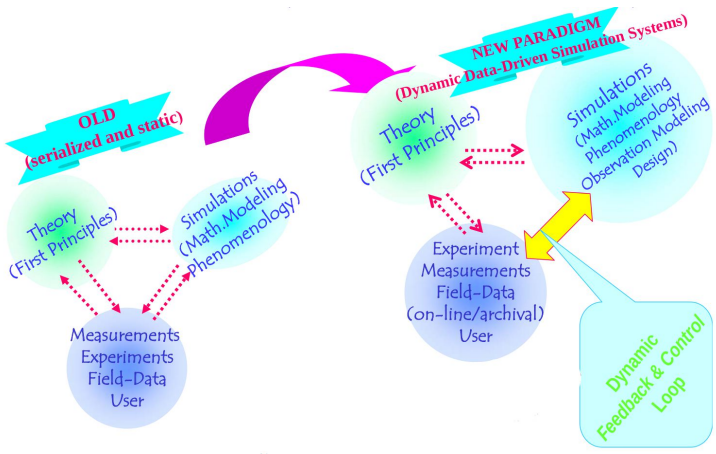

Figure 2.3: DDDAS scheme [Dar12b]

2.3.1 DDDAS benefits and challenges

As examples of computational science applications, the simulations are amongst the most well known. Simulations can be used to predict the weather and economy or to help prevent and manage critical situations such as fires and floods.

Regular simulations usually have databases filled with years of old data, and they are fed only by that old data. However, the simulations that are data-driven can be fed on-line with real-time data and with data that they are calculating on the run as well, thus allowing for better predictions and more accurate results.

If for some reason the application does not have the capacity to engulf all the data being received or if a more concise view of the problem is needed, the application also has the ability to select the data-sources, thus having the capacity to dynamically manage a compromise between system load and accuracy.

Aside from that, other benefits of DDDAS for simulations [Dar12a] are:

- Speed up the simulation through actual data usage since this avoids computations to produce data;
- Increase the simulation's accuracy by mitigating imprecise model definitions with related actual data;
- Enables real-time or near real-time modeling due to its dynamic capacity of receiving and tunning actual data;
- Improves measurements by selecting specific areas to monitor and dynamically activating/deactivating sensors.

This benefits however come with several problems and challenges that DDDAS applications must face [Dar12a], for instance:

- Complex application development involving experts from diverse areas;
- In order to (dynamically) incorporate algorithms from diverse areas, modularization

is a pressing requirement in these application's design;
- Devices generating data, and related protocols, are highly heterogeneous requiring strategies for uniform data management;
- Simulations that process large amounts of data usually require large amounts of computational power.

The middleware is particularly suited to solve challenges from this area, for it implements the concept of a session that aggregates information from several heterogeneous data-sources and dynamically disseminates it to large groups of clients sharing an interest.

2.3.2 Cloud computing and DDDAS

By deploying a DDDAS application into a Cloud, it immediately gains the benefits of scalability and elasticity. Scalability will allow the application to keep running while maintaining a required quality of service level, independent of system load, and paying only what is being used. Elasticity will allow the application to obtain more nodes in the scenario of a data peak, or to dismiss them should the system eventually become idle. Thus, if a DDDAS application requests more sources, the cloud provider can automatically give it more nodes, in order to manage the new amounts of data being received by those newly requested sources. Furthermore, the application and its clients can also take advantage from cloud's geographical distribution and ubiquitous access characteristics, previously mention in section 2.1. Thus, due to cloud's characteristics and the dynamism inherent to a DDDAS application, one can conclude that the two fit well together.

To build such an application, capable of solving the challenges from all the previously discussed areas, one needs a framework to aid with the integration of all the patterns and components required to solve and manage these problems. Such a framework is Apache Camel, with its long list of integration patterns and components, it allows for a faster development, which is also more modular and easy to maintain. Furthermore, this framework also allows aided with the integration of CEP systems into the middleware, namely Esper.

2.4 Apache Camel

2.4.1 What is Apache Camel?

Building complex systems often requires the integration of several components. These components usually have several technologies that all have their ways of dealing and processing data. Thus, gluing all the different components of a complex system in order for all their technologies to be able to communicate and operate seamlessly becomes a task hard to tackle [IA11].

Given the complexity of such a task, and how specific it is to each system, its components and the set of technologies it uses, many of the solutions are custom. This translates into a huge problem: such solutions are hard to explain to newcomers, they can only be

improved if one has deep knowledge about them and they cannot be applied to other similar problems. Consequently this results in a huge drag for software maintenance and updates.

To simplify this task, Gregor Hohpe and Bobby Woolf pioneered and developed the concept of Enterprise Integration Patterns (EIPs) [Myy12]. An EIP is the blueprint of a solution to an integration problem between several components, but not the solution itself. EIPs suggest ways to structure and organize code in order to make the implementation of the specific solution as easy and maintainable as possible.

Apache Camel is a framework that implements many of the solutions proposed by Gregor Hohpe and Bobby Woolf, resorting to languages like Java, XML, Groovy or Scala. Therefore, Apache Camel can be seen as a quick and out of the box way to solve integration problems between several components using different technologies.

2.4.2 Why use Apache Camel?

There are several advantages to using a framework to deal with component integration.

First, the programmer does not need to invest time on the details of the implementation and can therefore focus on the functionality of the application itself, thus speeding its development.

Second, because Camel well documented and also has a strong and active community (in both StackOverflow and Camel forums), it is easy to reach out for help and to learn the framework. This in turn, makes it easier for newcomers to enter projects because all they need to know is the framework and not the specific details of the project.

Last but not least, the two previous advantages result in projects easier to maintain and evolve over time, thus reducing adaptation costs of the team.

For the specific purpose of this project this framework also offers other advantages. It has a Java Domain Specific Language (DSL) that allows for auto-completion in the IDE, automatic type converting, and it also has extensive support for unit testing, which is a feature that lacks in other frameworks [IA11]. Furthermore, it is also a lightweight framework, quick to install and test, unlike the heavy Enterprise Service Buses (ESB) that although have more features, also have added complexity [Wah11] that is not needed for the size of this project.

2.4.3 Apache Camel's main concepts

In this section we describe Apache Camel's main components so the reader can have a better understanding of how Camel works and how it is practically applied to the project. The main building blocks are [IA11]:

Messages

A message is an object with a body, a header and additional fields. It contains information to be evaluated by processors and endpoints.

Exchanges

Exchanges are objects which contain two Messages: an *in* message and an *out* message. The *in* message is the message we send to our destination, and the *out* message is its reply to us. One can think of an Exchange as an envelope containing the message we wish to send, and that is then returned to us with the reply.

Endpoints

Endpoints are locations from which input data is consumed from and output data is sent to. Endpoints are configured by URIs with several options depending on their specific type. Because endpoints can consume or produce data they can be instantiated as a Producer or as a Consumer object, which behave differently depending on the type of endpoint that originated them.

Components

Components are factories of endpoints and they can be added to routes. This way, when the route is started through its Camel context, the components create the necessary endpoints, which then will be defined as consumers or producers depending on their role. Camel in Action presents the main types of components used, organized by functionality, which are:

1. File I/O: File, File Transfer Protocol (FTP);
2. Asynchronous messaging: Java Message Service (JMS);
3. Web services: CXF;
4. Networking: MINA;
5. Databases: Java Database Connectivity (JDBC), JPA;
6. In-memory messaging: Direct, Staged Event-Driven Architecture (SEDA), VM;
7. Automated tasks: Timer, Quartz.

However, there are many more, available through the official website and through communicates. The project uses the File, Esper, SEDA, HTTP, JMS and many other components. There are over 80 components available for use, so to fully understand them we recommend the official Camel documentation, which includes material made by the creators and by the community[5].

Processors

Processors are nodes capable of using, creating or modifying incoming exchanges. When routing messages from one endpoint to another, exchanges flow from processor to processor until they reach their final destination. If a route is a highway, then one can see processors as pit stops where the maintenance and work are done before getting back to the road.

Routes

A route is a collection of processors, organized into a graph. Exchanges originated

[5]Apache Camel Documentation: `https://camel.apache.org/documentation.html`

in producers flow from processor to processor inside that graph until they reach a consumer.

2.4.4 Apache Camel and the middleware

With all the advantages previously enumerated, Apache Camel is used as the main routing mechanism in the middleware. Thus, each time the middleware receives data, it redirects it to a Camel's component, which then transforms that message into a Camel message or exchange, and then forwards it through a route.

This allows for the middleware to be easier to decouple, and allows us to test new patterns as they are developed in order to improve the efficiency of the middleware.

However, another reason for using Apache Camel, is the previously referred Esper component. This component allows the middleware to effectively process CEPs, and Camel's integration with this component is also straightforward. CEP is important in that it allows the middleware to process large streams of data in a manageable way, and Esper is specially efficient at doing that, so the fact that Camel integrates this specific component well is important.

The main configuration options for the Esper component are the **eql** and the **pattern** statements, as can be seen:

Listing 2.1: Pattern statement

```
1  from("esper://cheese?pattern=every␣event=MyEvent(bar=5)")
2  .to("activemq:Foo");
```

Listing 2.2: EQL statement

```
1  from("esper://esper-dom?eql=insert␣into␣DomStream␣select␣*␣from␣org.w3c.dom.
       Document")
2  .to("log://esper-dom?level=INFO");
3  from("esper://esper-dom?eql=select␣childNodes␣from␣DomStream")
4  .to("mock:results");
```

Furthermore, the official documentation also refers to yet another extension for Camel, Camel Extra[6], which has additional components, and a working demo for the Esper component.

Although Apache Camel is a quick and easy way to solve interaction problems, efficiency problem may still exist. These problems may come from the fact that certain components do not work as expected, or may be external and depend on the network traffic conditions or even on the processing power of the sources themselves.

To identify such problems we used simple metrics to measure how well the software was behaving and profiling tools to identify problematic areas in the middleware. Therefore we end this chapter with the final section, which introduces the reader to these concepts and tools.

[6] http://code.google.com/a/apache-extras.org/p/camel-extra/?redir=1

2.5 Metrics and Profiling

In order to take the right decisions we first needed to identify the problems and then fix them. To help us identify these problems we resorted to a set of metrics and to a profiling tool. Therefore, in order to better understand the work developed, it is important to have a basic notion of some metrics used to measure how well a software is behaving and to know what tools are available to test that. This chapter presents this reader with these basic notions and then presents several tools which he can also use to test and profile other Java applications.

2.5.1 Concepts

Here we present the main concepts that should be known when analyzing performance. These metrics have been used by the community for years, and they provide an insight on whether the changes made to software are worth the costs or not, or if it changed for better at all.

Speedup

Used in parallel computing, the term speedup refers to how much a parallel version of an algorithm is faster than its sequential counterpart. The most common mathematical definition of speedup is the formula:

$$S_p = \frac{T_1}{T_p} \tag{2.1}$$

Where S is the speedup, p is the number of processors used, and T is the time that the algorithm takes to run, with p processors. Optimal speedup, (also known as Linear speedup) happens when $S_p = p$, however, thanks to communication, synchronization and other additional costs, this is rarely achieved.

The term speedup also has other more recent definitions [Ert94] that encompass the variation in the speedup, however for the purposes of this thesis, the former term is the one used.

When discussing speedup it is also important to refer to Amdahl's law [Amd67], which states that the speedup of a program using multiple processors is limited by the sequential portion of the problem being undertaken. If the execution time of the sequential part of a problem is X seconds, then no matter how many processors we use, we will never be able to execute the parallel program in less than X seconds. Thus, the maximum speedup is limited by the sequential part of a problem.

This however can be countered by Gustafson's law [Gus88], which states that by increasing the number of processors a program uses, the program becomes able to tackle more and more data. Thus, if the complexity of the sequential section of the problem is smaller than the complexity of the section that can be parallelized, and if there is enough data to process, one will eventually reach an infinite speedup.

Efficiency

Efficiency is uses the speedup metric and it is used to know how much processing power is being wasted in communication and other tasks not related to processing data. This metric typically is between zero and one, and its mathematical formula is the following:

$$E_p = \frac{S_p}{p} \tag{2.2}$$

Where E is the efficiency, p is the number of processors used, and S_p is the speedup gained.

Latency

Latency has different meaning depending on the area it is used. For example, [Var95] defines latency as the time it takes to complete a certain task, while in the networks domain, latency represents the delay of a packet, this is, the time it takes a packet to go from location A to location B [RB06].

For the purpose of this thesis, we consider both definitions, properly identified when used. It is important to notice however, that the main contributors for this type of latency are the propagation, transmission router and computer processing delays, which were not evaluated specifically - the only measurement considered important was the travel time, which technically encompasses all of the above delays.

Throughput

Generally, throughput is the amount of work done by a computer in a certain amount of time [Var95; RB05]. In the networks domain is the rate of successful message delivery over a communication channel, during a certain amount of time. This is usually measured in bits per second [Thr].

For this thesis however, the used definition will be the first one, and it will be measured as the amount of processed messages per minute.

Scalability

At the time of writing of this thesis, there is no consensus on the formal definition of scalability [Hil04]. However,there are still some good definitions that can be used, such as the one proposed by [Pac11], which defends that "A program is scalable if the problem size can be increased at a rate so that the efficiency does not decrease as the number of processes increase." Hence, this thesis adopts this definition of scalability.

For the specific purposes of the middleware, this means that no matter the amount of messages the middleware has to process, if the filter expressions being used do not change, the latency as defined by [Var95] should not increase.

2.5.2 Profiling tools

Before making the middleware scale out, it is important to make sure we are using all the possible resources in the current machine. The only way to find that out is by resorting to profilers. Profilers are tools that allow the programmers to check CPU, Memory, IO, Network, etc, and to make decisions based on those values. For example, if our

middleware never surpasses the 20% CPU usage mark, then there is no need to ask for another machine, because the current being under utilized.

To achieve this purpose, we considered a wide range of Java profiling tools. These tools range from simple tools that give us basic information to more complex tools that would require changes in the code but that are more powerful. As a thumb rule, the more powerful a given profiler is, the more linked to the project it becomes. This means changing the project code in certain areas or even hard coding the required tests in the methods we wish to measure.

For this project we considered the following set of tools [Ske11]:

JMap [7]

The most basic of tools considered, JMap comes out of the box with the Java Virtual Machine (JVM). It allows the creation of histograms of the JVM and it is a quick tool. However, it causes the JVM to halt execution if there is too much data to collect. Furthermore, this is only a command line tool and has no graphics at all, thus the information becomes hard to read.

VisualVM [8]

The selected tool used to profile the middleware. A more advanced tool, with graphical information and added functionality, VisualVM also comes out of the box with the JVM. VisualVM is a combination of several tools, that allows the programmer to collect information about CPU usage, Memory usage (both heap memory and perm memory), actual number of threads, actual number of created classes, objects and how much resources those entities are currently consuming.

It can be launched either before or after the Java application, and although some of its functions actually halt the JVM, the majority of them do not. In some cases it does make the Java application slower, but in our case that was not a problem.

BTrace [9]

BTrace is the step after VisualVM. While VisualVM simply collects everything, BTrace allows the programmers to create special scripts with annotations that detail what to retrieve and what to ignore. The problem with this alternative is that it requires changes to specific sections of the already existing code and it does not come out of the box with the JVM, like the two previous approaches.

Other third party tools

There are also other third party tools like JProfiler[10], JVM Monitor[11] (for Eclipse), and many others on the java-source.net[12] repositories. These tools are all capable of profiling Java applications one way or another, with different levels of detail. Because comparing all these tools would take too much time, and because all we

[7] http://docs.oracle.com/javase/6/docs/technotes/tools/share/jmap.html
[8] http://visualvm.java.net/
[9] https://kenai.com/projects/btrace
[10] https://www.ej-technologies.com/products/jprofiler/overview.html
[11] http://www.jvmmonitor.org/
[12] http://java-source.net/open-source/profilers

wanted is already done with VisualVM, we decided to use that tool instead.

2.5.3 Profiling in the cloud

Because resources usage is usually linked with the price customers end up paying, profiling in the cloud is different from profiling the applications locally.

To make this possible, each cloud provider usually has its own set of profiling tools, that enable customers to check the usage of their machines, and to spawn more machines or to kill them if necessary. However, these features are usually hidden behind other functionalities.

In Google App Engine all the profiling can be accessed in the project's dashboard, however to retrieve more precise information the customer must enable the billing options. This allows the user to see how the quotes are being used depending on each service. If the quotas expire, billing is applied.

In AWS, the profiling is hidden behind an alarm system, CloudWatch[13]. The customer can set up alarms that will notify him via e-mail if a certain machine is using too much resources or if the system needs to scale out or up. This alarm system is also used as a base to the auto scaling system[14], which allows customers to decide when to create or kill machines. This simple system allows the customers to monitor CPU utilization, network bandwidth and I/O reads and writes.

With the basic notions of metrics and profiling established, the next chapter presents the reader with an evaluation of the middleware and how it works from a high level point of view.

[13] https://aws.amazon.com/cloudwatch/
[14] AWS Auto Scale - https://aws.amazon.com/autoscaling/

3

The Middleware and its Evaluation

In this chapter we present a general overview of the middleware being evaluated, from the session concept it offers, to its concrete implementation. From there we proceed to the analysis conducted in the scope of this thesis, which assesses the middleware's performance and scalability.

3.1 The Middleware

A distinguishing characteristic of the middleware is the concept of shared interaction context that enables multiple clients, with similar interests, to conjointly access multiple, possibly heterogeneous, data sources. This context, called a session, can also be dynamically changed according to the needs of the users, thus allowing for more flexibility in sharing data.

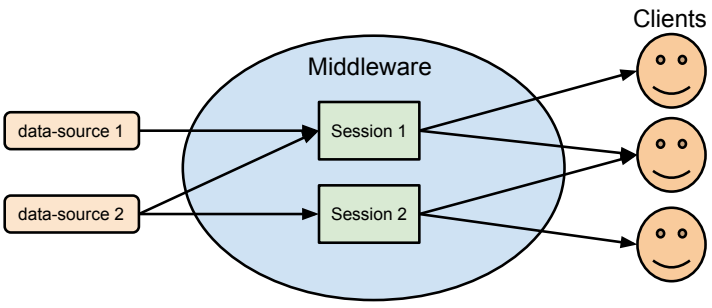

Figure 3.1: First view of the middleware

3.1.1 Session Abstraction

In the context of this thesis a session is a concept that encapsulates the interaction model between a group of clients that shares a set of interests, the creation of that session and the sources. As depicted in Figure 3.2, a session is characterized by:

Session id - uniquely identifies the session, allowing other clients to connect to it;

Users - set of clients that are currently connected to the session;

Owner - the user that created the session. He/she is the only with access privileges to execute administrative operations and should also be responsible for the session's costs in the cloud;

Interaction Model - dictates how the clients within this session receive data. Currently, three models are supported: *publisher-subscriber*, *producer-consumer* and *streaming*;

Dynamic reconfigurations - allows the owner of a session to change the latter's base interaction model at any time. It also allows clients within a session to adjust the session's interaction model to their specific needs;

Heterogeneous data-sources - a session is able to retrieve data from multiple heterogeneous sources, namely XMPP, RSS, HTTP and Twitter.

CEP aggregation functions - defines a set of rules for data filtering and aggregation. These rules are then applied to the session's incoming data. Users have the power to change these rules at any time. If the user demanding this reconfiguration is the session's owner, then the changes have a session-wide impact, i.e., they are directly applied to the data received by the session and affect all users within the session. On the other hand, if the user is a client, then the changes are applied to the data after it has been filtered by the session's conditions and it only affects that client.

Repository - a session may be equipped with a repository where it stores the raw data it receives from the data-sources, the result of the CEP aggregation and processing functions, and the dynamic reconfigurations applied during the session's execution. This feature allows the session to be replayed in the future, for optimization, auditing or any other purposes;

Status - each session has one of six states: *new, starting, started, paused, stopping* and *ended*. These states define the session's life cycle and are saved in the repository, so the session can later on be replayed with them as well.

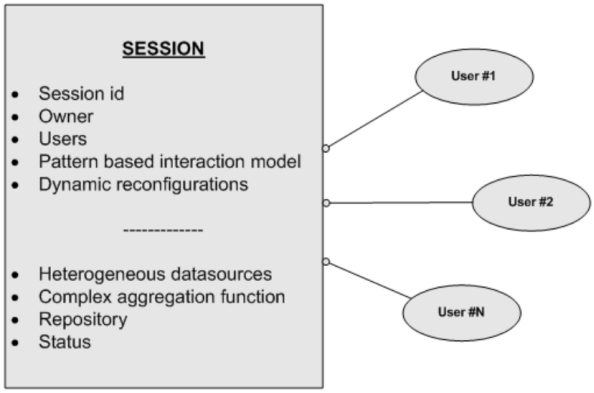

Figure 3.2: Session abstraction [Dom13]

The concept of a session abstraction offers several advantages. For example, by grouping clients based on their interests, a session naturally provides a more efficient way of retrieving data. It eliminates the need for each client to query data-sources individually, thus decreasing the risk of flooding it with too many requests. Another advantage is that a client may self-adjust the session's global interaction model and further refine (filter) the processed data-stream requirements, both functional or non-functional (such as having increased battery lifetime or a less network traffic). Furthermore, the client can change his personal interaction model at any time, depending on the model being used by the session, without having to re-establish the connection to the service. These features effectively aid clients with a weak connectivity or processing power by allowing them to change the amounts of information they receive and process at any time. A final advantage of this concept, is that all technology-aware interactions with the defined data-sources are performed by the middleware. This has several advantages for the end-user, namely in the seamless integration of new data-sources, which in turn removes that extra complexity and allows an easier development of client applications.

Deploying a session abstraction in a cloud, offers even more advantages than the aforementioned ones. Cloud's services are also virtually unlimited depending only on the price we are willing to pay for it, allowing virtually unlimited storage space for the repository and computational resources to perform the computations upon the datastreams.

Furthermore, as mentioned in section 2.3.2 clouds also allow the dynamic recruitment and dismissal of workers, resulting in a steady level of performance independently of peaks. Reliability is also an important feature, since clouds usually have distributed systems to ensure that the service is always up, thus promoting an ubiquitous access which in turn allows the session abstraction to be useful to a wide range of devices such as smart-phones, tablets and others.

Another concept that the abstraction of session naturally explores is also co-location, previously defined in Section 2.2.2.4. Provided that a session middleware is implemented within a cloud, then all clients within the session will benefit from co-location.

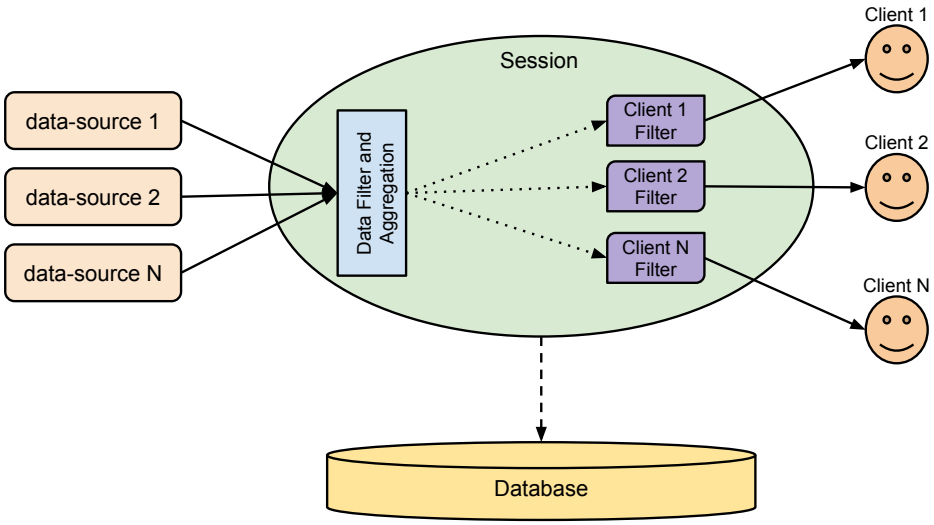

Figure 3.3: Example of a Session

3.1.2 Middleware Architecture

This section presents the reader with a detailed description of the middleware's architecture. We begin by giving a general architectural overview limited to the application's components and how they interact. Then we focus our attention to the internals of these components, where we introduce the reader to the core of the application, to its layers and to the interactions between them.

3.1.2.1 General Architecture

An eagle's eye of the application can be seen as dividing it into three main components:

- Datasource Interface
- Middleware Core
- Client Interface

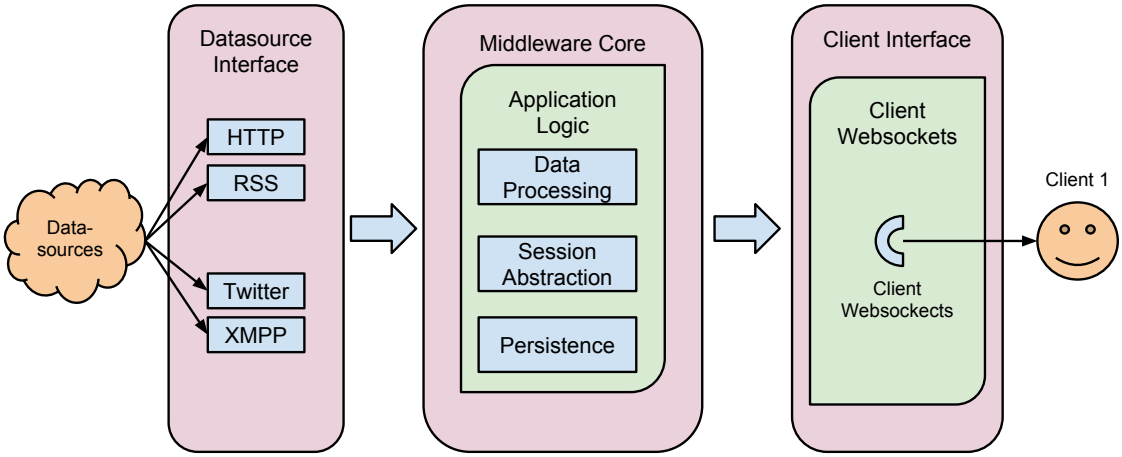

Figure 3.4: Generic Middleware Architecture

The data-source interface is an adaptation layer for handling the multiple types of data-sources, where type is bound to the technological framework required to interact with such source. Every type of data-source requires its own adapter that is responsible for relaying the incoming data to the core of the middleware. Currently, it supports the protocols of XMPP, Twitter and RSS, and it can also use Xpath queries to retrieve information from HTML pages. This component serves as an integration layer so the middleware core can then process the received messages, oblivious of the specificities of the data-source's communication protocols.

The Core component is where the logic of the middleware is implemented. It processes and routes the messages to the correct sessions, subsequently processing them according to target session's configurations and delivering them to the client interface. The Core is also responsible for all dynamic behavior of the program, as well as for the persistence of the stored data.

Finally, the client interface is the component responsible for sending the information to each client according to the client-specific interaction model. This communication is Web based, built on top of websockets.

3.1.2.2 Specific Architecture

The heart of the application is the middleware core component. To achieve the level of scalability and efficiency the core uses two main technologies - Apache Camel and Esper. Apache Camel is a routing system for messages (see Section 2.4) and Esper is a component used for complex event processing (see Section 2.2.1.2), which allows the middleware to process streams of events in an efficient way.

These two technologies are present in all of the three layers composing the core:

- Datasource Messaging Layer

- Session Messaging Layer

- Client Messaging Layer

The middleware core was built to handle several data-sources, sessions and clients, as presented in Figure 3.1. However, considering that this is the first exposure of the reader to the core component, explaining a realistic scenario with many data-sources, sessions or clients would be complex and hard to understand. Thus, we explain how the core component works by illustrating a very simple scenario with only one data-source, one session and one client, and we walk the reader through every step of the process where we explain the path that a message takes when inside the core component, as illustrated by Figure 3.5.

We start with the Data-Source Messaging Layer, which is responsible for the camel route (R1) that filters and processes messages from the Datasource interface. This route then sends them to an Esper endpoint (E1) depending on the session. In the Session Messaging Layer, another camel route (R2) picks up the messages left in E1, processes them according to the session configurations and then sends them to a final Esper endpoint (E2). The final camel route (R3) then picks the messages from E2 and sends to clients by via websockets. This last route is part of the Client Messaging Layer.

These three layers work together to create, update and manage a session container, which holds information about each session and is then persisted in the repository, as can be seen in the Figure 3.5.

It is this repository that saves all the data processed by the session, as well as all the session's states and information about any other events that may have occurred within the session. The storage of this information, that works as a history tracker of activities, then allows the users to replay the session as it was.

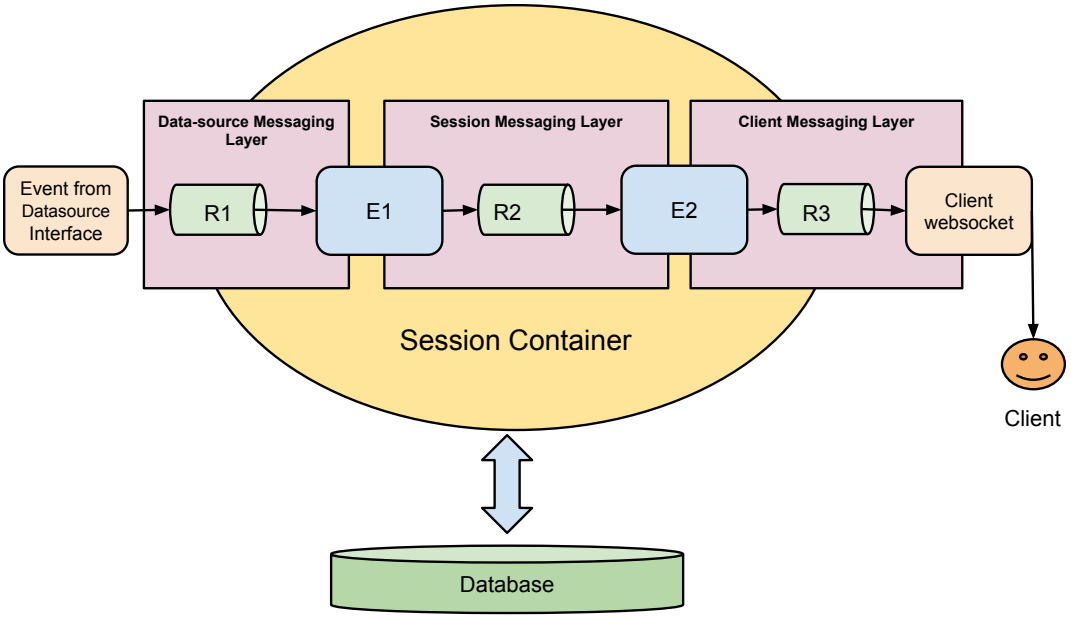

Figure 3.5: Middleware Core Architecture

3.2 Evaluation

This section focuses on the evaluation of the middleware, in its exploration how it can be improved. Thus, we start by with a performance analysis in order to determine if our suspicions about poor performance are correct, and then we follow with an overview of the data layer and the problems it may pose regarding efficiency and scalability.

3.2.1 Performance

In real life situations, the middleware needs to be scalable and elastic to deal with the dynamic requirements of the clients, and with the fluctuations not just in their numbers, but also in the numbers of active sessions and data-sources.

However, the solution currently implemented is only scales up (meaning it has to be deployed in a better machine to handle more work) and does not scale out (meaning it can make use of additional machines in order to produce more work). Because scaling up is not possible in real time without stopping the middleware and deploying it in a stronger machine, we are stuck with the current setup. Therefore, we have reasons to believe this will have performance problems when under stress.

To test this theory, we stressed the server by performing a battery of tests which increased the number of sessions, and we measured the total execution time that it takes for the middleware to process the received data. Then we analyzed the results and took conclusions based on them. Therefore, the next subsubsection describes the details of the platform created to perform the tests as well as the tests themselves. Then we introduce

the reader to the experimental setup used to run and make the analysis, followed by the presentation of the results obtained. Lastly, we conclude with the analysis, from the performance and operational perspectives, of the collected results.

3.2.1.1 Tests and platform description

In this section we describe the tests and make an introduction to the platform used to run those tests. The created tests stress the system by varying the following parameters:

- Number of sessions;
- Data-Source message rate per minute;
- Esper expression to process the data at session and client level.

The number of sessions listening to a data-source impacts directly on the number of Camel routes inside the middleware and, subsequently on the amount of work done. As such, our tests included 1, 2 and 4 sessions to see how far we could push it.

We are also increased the number of messages sent per minute by each data-source in use, in order to stress the middleware with large amounts of data, feasible in a real life scenario. The number of messages sent per minute, per data-source, was 60, 120, 240, 480, 1000, 2000 and 4000.

Finally, the Esper expressions used to filter the data according to the session's configurations could also have a significant impact in the CPU needed and could be the reason for the loss in performance. To check this possibility we tested three Esper expressions:

- Exp0, which simply lets everything pass;

Listing 3.1: Exp0

```
1 select * from pattern [every e=net.jnd.thesis.domain.Event]
```

- Exp6 and Exp7 which calculate the average of all the results received in the last 30 and 60 seconds respectively.

Listing 3.2: Exp6

```
1 select avg(cast(value, float)) as exp from net.jnd.thesis.domain.
      Event.win:time(30 sec)
```

Listing 3.3: Exp7

```
1 select avg(cast(value, float)) as exp from net.jnd.thesis.domain.
      Event.win:time(60 sec)
```

To run the tests we deployed an artificial data generator to simulate a data-source, the middleware filtering the data using the previously specified esper expressions, and a client to which the middleware sent the output of the computations. With this in mind,

our metrics focused on measuring the middleware's overall execution time in seconds. To effectively perform the stress tests however, there were two main limitations that we had to consider: the number of messages a data-source can send, and the number of clients.

The first limitation can be addressed by using an event-driven protocol, such as XMPP or Twitter. For the purposes of the tests we decided to use XMPP. However, even though the middleware supports this protocol, we cannot use Google's service - which was used in previous functional evaluations - to stress the application because our accounts will be put on hold if we start sending to many messages in a short amount of time.

To workaround this limitation, we installed our own XMPP server, by setting up Openfire[1] and then adapting the middleware so it would receive message from Openfire and not from Google's service. Openfire allowed us to have a local XMPP server, customized to the needs of the project and that does not block users when they send too many messages. Furthermore, because it has a considerable amount of plug-ins it also allows us to extend its basic functionality, for instance to enable seeing the history of messages traded between two accounts, which revealed to be important for debugging purposes.

The second limitation, the number of clients, was surpassed using two different browsers. However we realized this was not enough to actually stress the system, so after an extensive research we considered using a load testing tool such as Tsung[2]. Tsung would allows us to simulate hundreds of clients as well as their actions with the system, unfortunately, due lack of time, Tsung was not used.

With the previous limitations surpassed, several Java applications were created to stress the middleware. These applications had two main functions:

- Stress the middleware by simulating sensors that send data at very fast rates;
- Evaluate the logs produced by the middleware in order to organize information and take conclusions.

The first set of applications uses the Jabber Smack API[3] and it allows us to simulate various sensors (data-sources), and to send thousands of messages in short amounts of time using the XMPP protocol. Therefore, when a sensor wants to communicate with the middleware, it sends a message to a specific e-mail that the middleware is listening to, in this case *thesis.dimfccs@gmail.com*.

The second set of applications was created to evaluate the content of the logs generated by the execution of the middleware. Since the logs are usually thousands of lines long, it becomes very hard and time consuming to read those logs and take conclusions from them. The applications created perform a series of functions, like calculating the average amount of time a message spends inside the middleware, the middleware's execution time, throughput, and other useful information in a quick and effective way.

[1] http://www.igniterealtime.org/projects/openfire/
[2] http://tsung.erlang-projects.org/
[3] http://www.igniterealtime.org/projects/smack/

As an additional note however, it is also important to mention that all the tests were run using the debug mode provided by log4j in the middleware. This execution mode prints information about threads, execution times and other aspects of the middleware to a log file that is usually thousands of lines long. This can slow the execution when compared to the *off* mode, which simply runs the middleware at full speed.

Furthermore, besides the Java applications created, the execution of each configuration was also monitored using the jvisualVM[4] tool. This tool allowed us to monitor the CPU being used, threads created, memory and heap being used, and it also allowed us to profile which classes were consuming more resources.

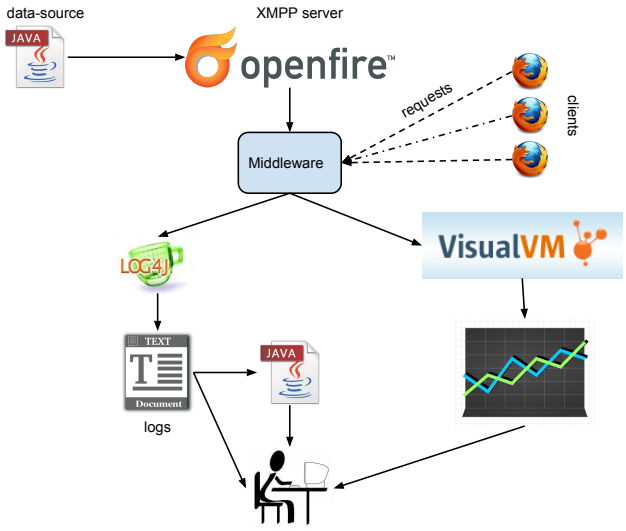

Figure 3.6: Architecture of the test platform created

3.2.1.2 Assessing the overhead of the Debug mode

To compare the cost that using the debug mode had, we conducted a small battery of tests, using the scenario with 1 source, 1 session and 1 client, sending 4000 messages per minute and using the Esper Exp7 expression. These simple tests compare the execution time of the middleware against the execution time of the data-source, which is always 60 seconds. We considered that the middleware started its execution when it received the first message, and ended it after delivering the last message.

The results of the tests are as follows:

[4] http://docs.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html

1 Source, 1 Session, 1 Client, Exp7, Debug on			
Message rate per minute	**Middleware execution time (s)**	**Delay (s)**	**Average delay (s)**
4000.00	647.00	587.00	608.33
	677.00	617.00	
	681.00	621.00	

Table 3.1: Table with the execution time of the middleware using Exp7 and with debug modes enabled

1 Source, 1 Session, 1 Client, Exp7, Debug off			
Message rate per minute	**Middleware execution time (s)**	**Delay (s)**	**Average delay (s)**
4000.00	594.00	534.00	521.67
	587.00	527.00	
	564.00	504.00	

Table 3.2: Table with the execution time of the middleware using Exp7 and with debug modes disabled

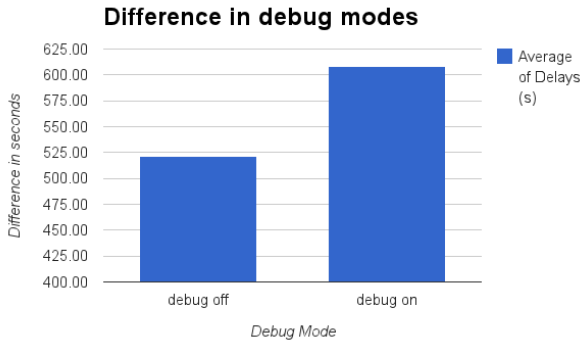

Figure 3.7: Difference of the delay in seconds between running the same test with the debug mode enabled and disabled.

As can be seen the Tables 3.1 and 3.2, the difference amounts to 86.66 seconds, which is roughly equivalent to 14.25%. Although this difference demonstrates the impact of the debug mode, which can be observed in Figure 3.7, it is not significant enough to invalidate the results obtained.

3.2.1.3 Experimental setup

This section presents the details of the hardware and software layers used to execute the testing platform. Depending on future releases of the programs and on different testing machines, adaptations may be required.

Hardware information:

- Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz
- 8GB RAM

Software information:

- Java version "1.7.0_51", Java(TM) SE Runtime Environment (build 1.7.0_51-b13), Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
- Openfire 3.8.2
- Jetty 8.1.14
- Linux Mint 15 Olivia
- Kernel version 3.8.0-19-generic (buildd@allspice) (gcc version 4.7.3 (Ubuntu/Linaro 4.7.3-1ubuntu1)) #30-Ubuntu SMP Wed May 1 16:35:23 UTC 2013

3.2.1.4 Experimental Results

Because the amount of tests and results gathered is massive, in this section we only show a representative portion of that data. The complete list of screenshots and charts can be consulted in Appendix A, annexed to this document. For each of the considered scenarios we present the tables with all the information on execution times and averages, a chart that compares that data together, and finally, the information collected from jvisualVM only for the heaviest case with 4000 messages per minute for all Esper expressions. Therefore, we present the test results for the following scenarios:

- 1 source, 1 session, 1 client;
- 1 source, 2 sessions, 1 client;
- 1 source, 4 sessions, 1 client;

The results include graphics that illustrate the amount of CPU and memory used as well as the number of threads and classes. We also present tables that expose the delay that the middleware took when processing the received messages when compared to the time that the data-source took to send them. For all the tested scenarios, the data-source always runs during sixty seconds, and the delay as well as the average delay are therefore the difference and the average of differences between the running time of the middleware and those sixty seconds.

1 source, 1 session, 1 client

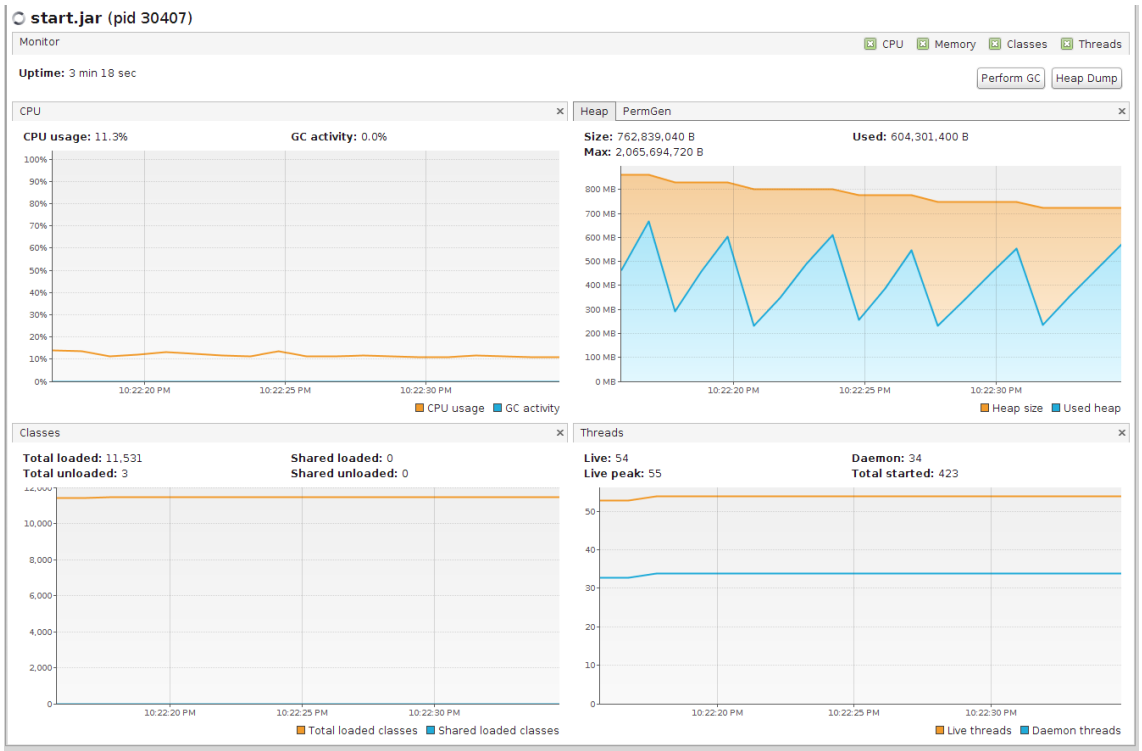

Figure 3.8: Resources used when running Exp0 with 4000 messages per minute

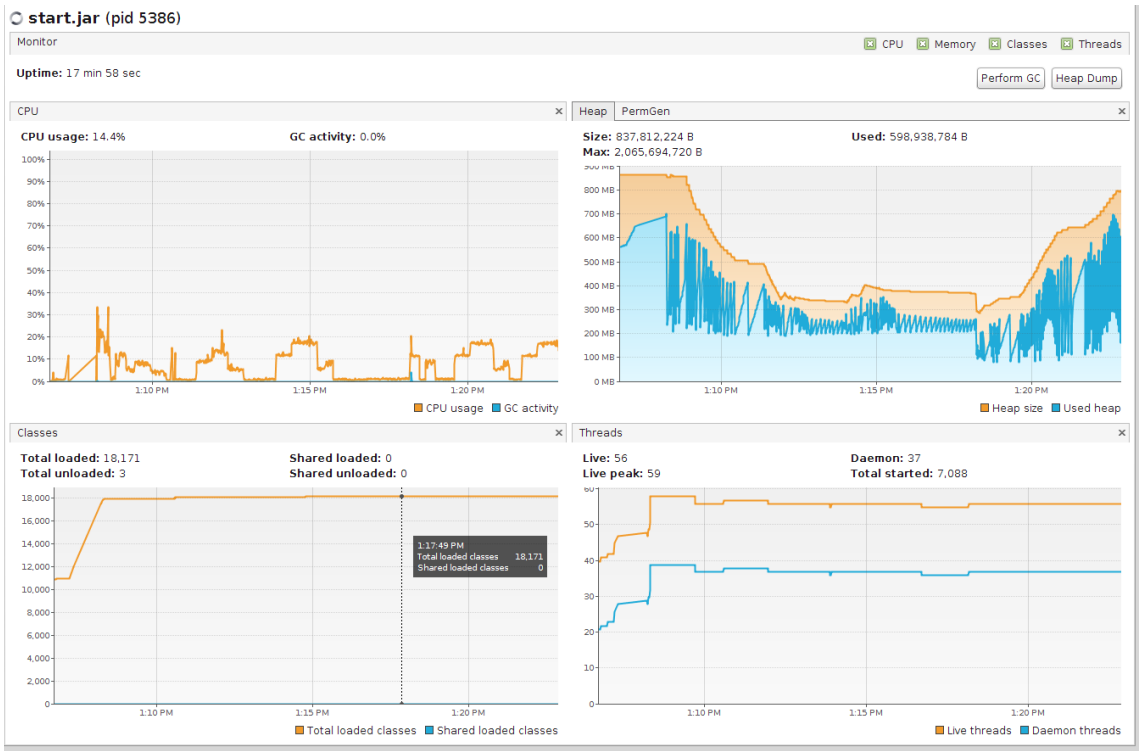

Figure 3.9: Resources used when running Exp6 with 4000 messages per minute

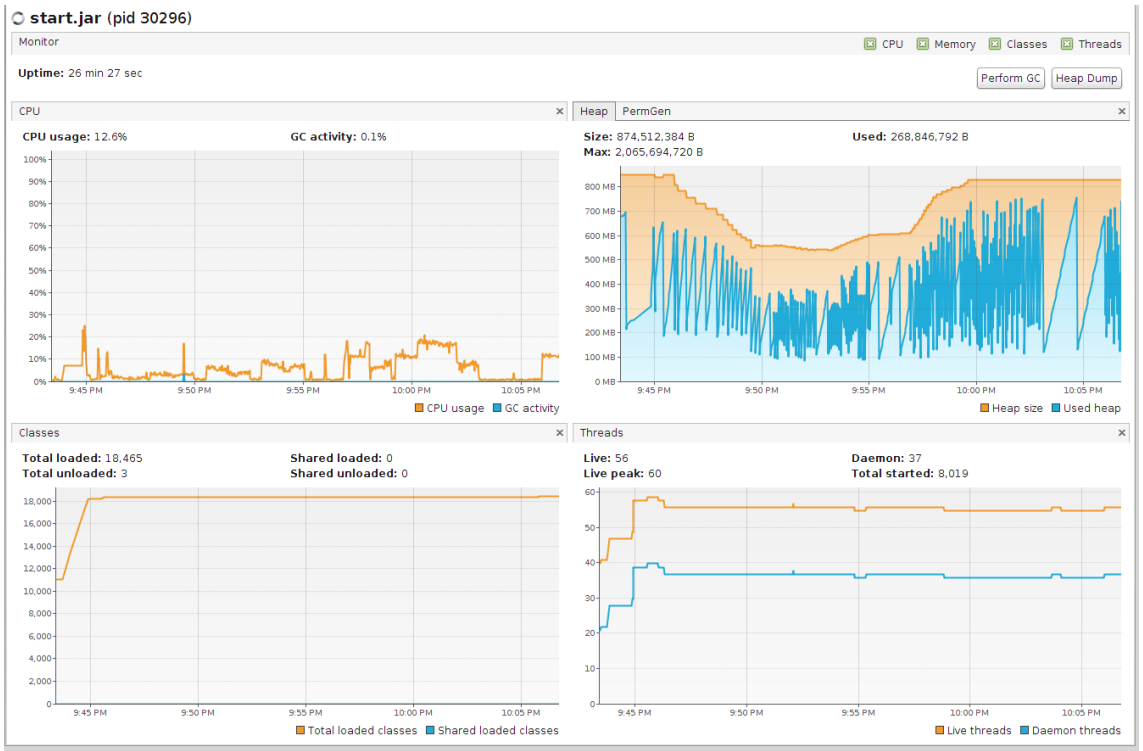

Figure 3.10: Resources used when running Exp7 with 4000 messages per minute

1 Source, 1 Session, 1 Client, Exp0			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	60	0	0.00
	60	0	
	60	0	
120	60	0	0.00
	60	0	
	60	0	
240	60	0	0.00
	60	0	
	60	0	
480	69	9	16.67
	69	9	
	92	32	
1000	159	99	109.67
	173	113	
	177	117	
2000	391	331	311.00
	383	323	
	339	279	
4000	561	501	512.33
	631	501	
	595	535	

1 Source, 1 Session, 1 Client, Exp6			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	88	28	28.00
	88	28	
	88	28	
120	89	29	29.00
	89	29	
	89	29	
240	89	29	29.00
	89	29	
	89	29	
480	108	48	49.00
	109	49	
	110	50	
1000	188	128	131.00
	188	128	
	197	137	
2000	316	256	284.00
	361	301	
	355	295	
4000	599	539	542.33
	617	557	
	591	531	

1 Source, 1 Session, 1 Client, Exp7			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	118	58	57.67
	117	57	
	118	58	
120	119	59	59.00
	119	59	
	119	59	
240	119	59	59.00
	119	59	
	119	59	
480	142	82	85.67
	147	87	
	148	88	
1000	245	185	183.63
	243	183	
	242	182	
2000	386	326	313.67
	363	303	
	372	312	
4000	647	587	608.33
	677	617	
	681	621	

Table 3.3: 1 source, 1 session, 1 client middleware execution times

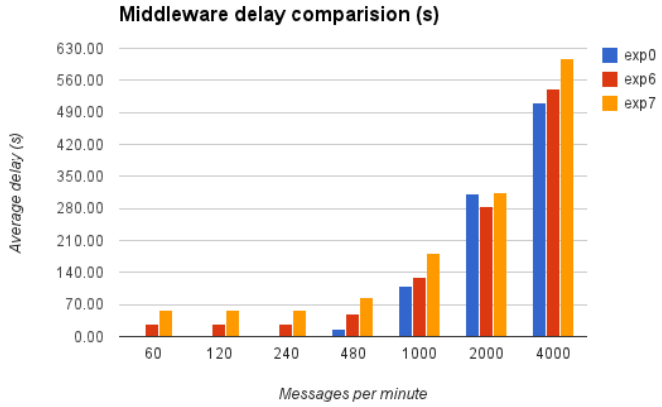

Figure 3.11: Middleware delay comparision

1 source, 2 sessions, 1 client

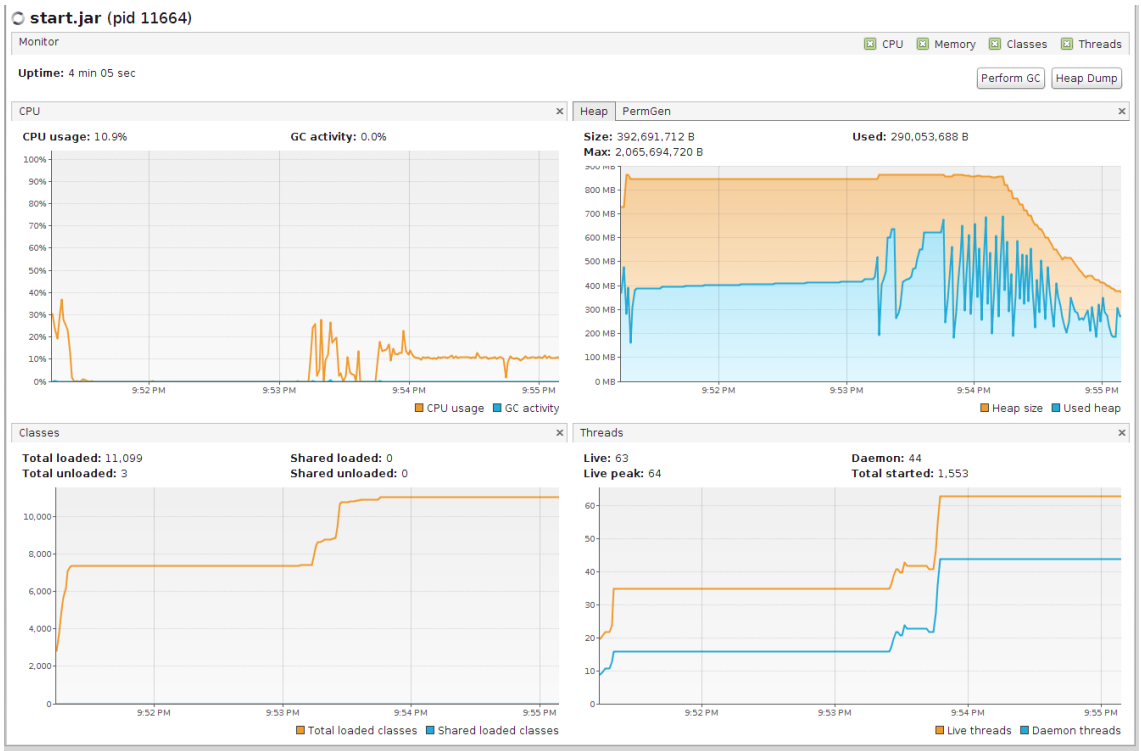

Figure 3.12: Resources used when running Exp0 with 4000 messages per minute

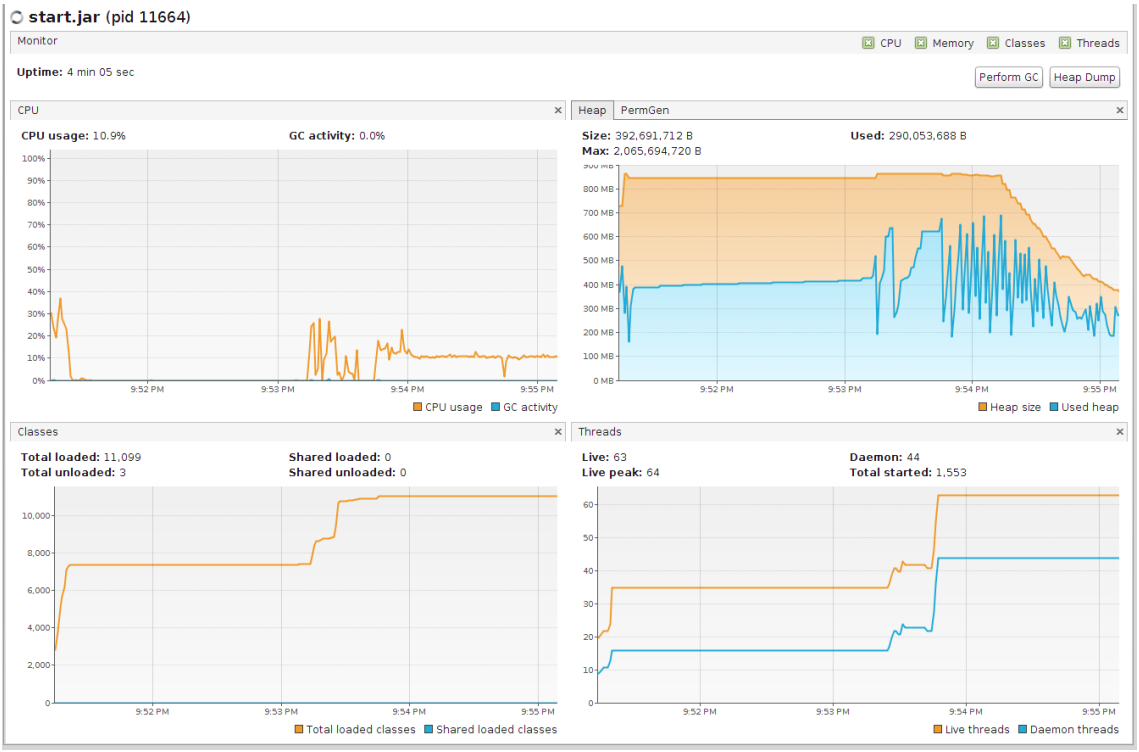

Figure 3.13: Resources used when running Exp6 with 4000 messages per minute

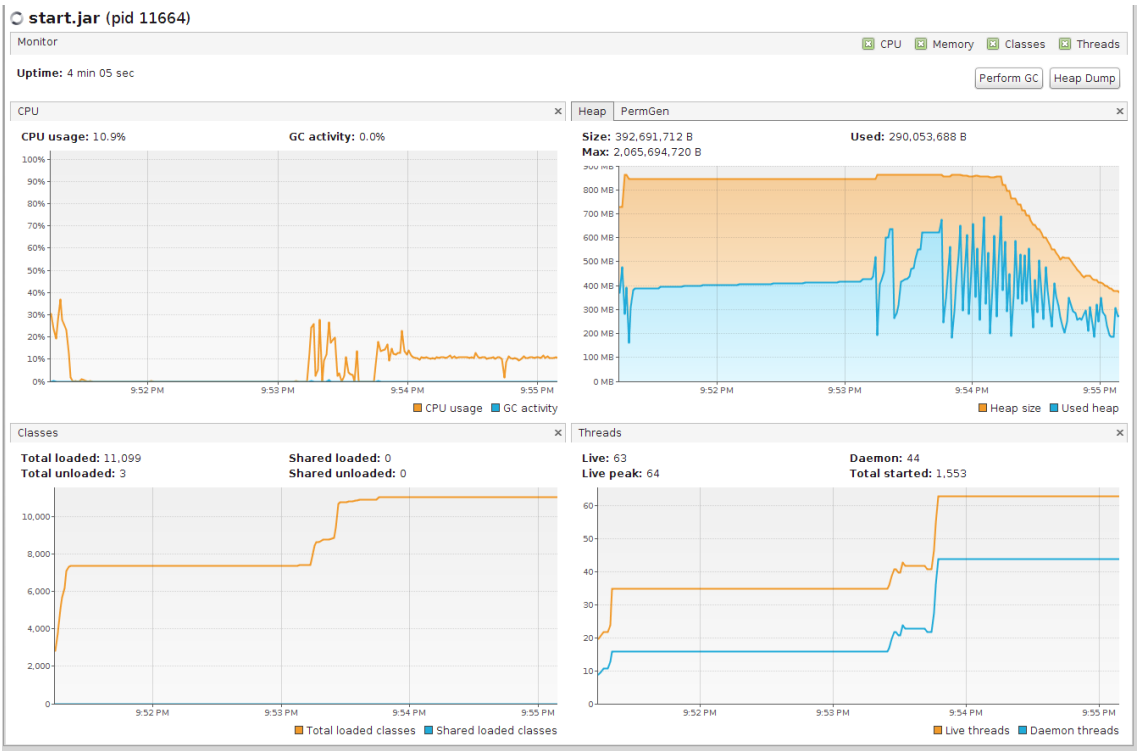

Figure 3.14: Resources used when running Exp7 with 4000 messages per minute

1 Source, 2 Sessions, 1 Client, Exp0			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	60	0	0.00
	60	0	
	60	0	
120	60	0	0.00
	60	0	
	60	0	
240	60	0	0.00
	60	0	
	60	0	
480	94	34	37.33
	98	38	
	100	40	
1000	211	151	147.67
	205	145	
	207	147	
2000	442	382	369.33
	425	365	
	421	361	
4000	1099	1039	961.00
	915	1039	
	865	805	

1 Source, 2 Sessions, 1 Client, Exp6			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	88	28	27.33
	87	27	
	87	27	
120	88	28	28.33
	89	29	
	88	28	
240	89	29	29.00
	89	29	
	89	29	
480	124	64	62.00
	118	58	
	124	64	
1000	233	173	174.00
	234	174	
	235	175	
2000	419	359	364.67
	429	369	
	426	366	
4000	769	709	728.33
	754	709	
	827	767	

1 Source, 2 Sessions, 1 Client, Exp7			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	118	58	57.67
	117	57	
	118	58	
120	119	59	59.00
	119	59	
	119	59	
240	119	59	59.00
	119	59	
	119	59	
480	146	86	85.00
	148	88	
	141	81	
1000	243	183	179.67
	234	174	
	242	182	
2000	418	358	364.33
	435	375	
	420	360	
4000	746	686	708.67
	780	720	
	780	720	

Table 3.4: 1 source, 2 sessions, 1 client middleware execution times

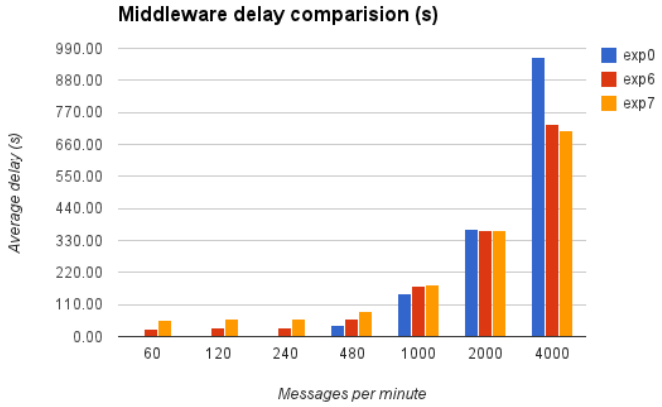

Figure 3.15: Middleware delay comparision

1 source, 4 sessions, 1 client

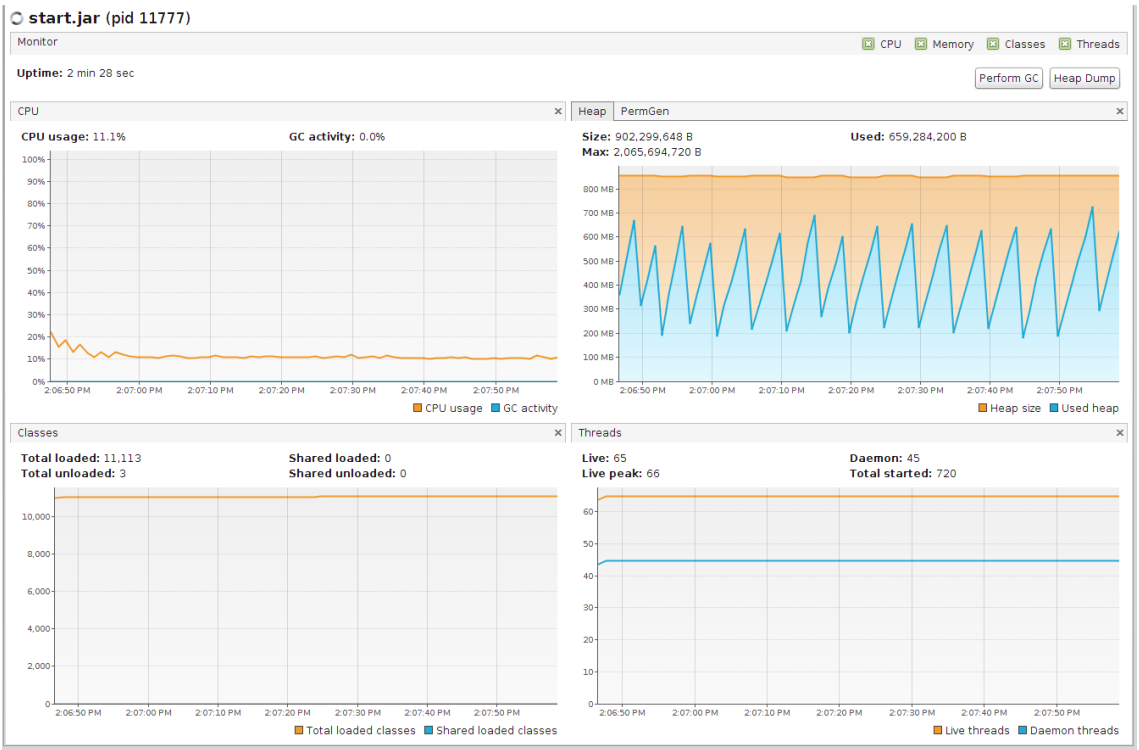

Figure 3.16: Resources used when running Exp0 with 4000 messages per minute

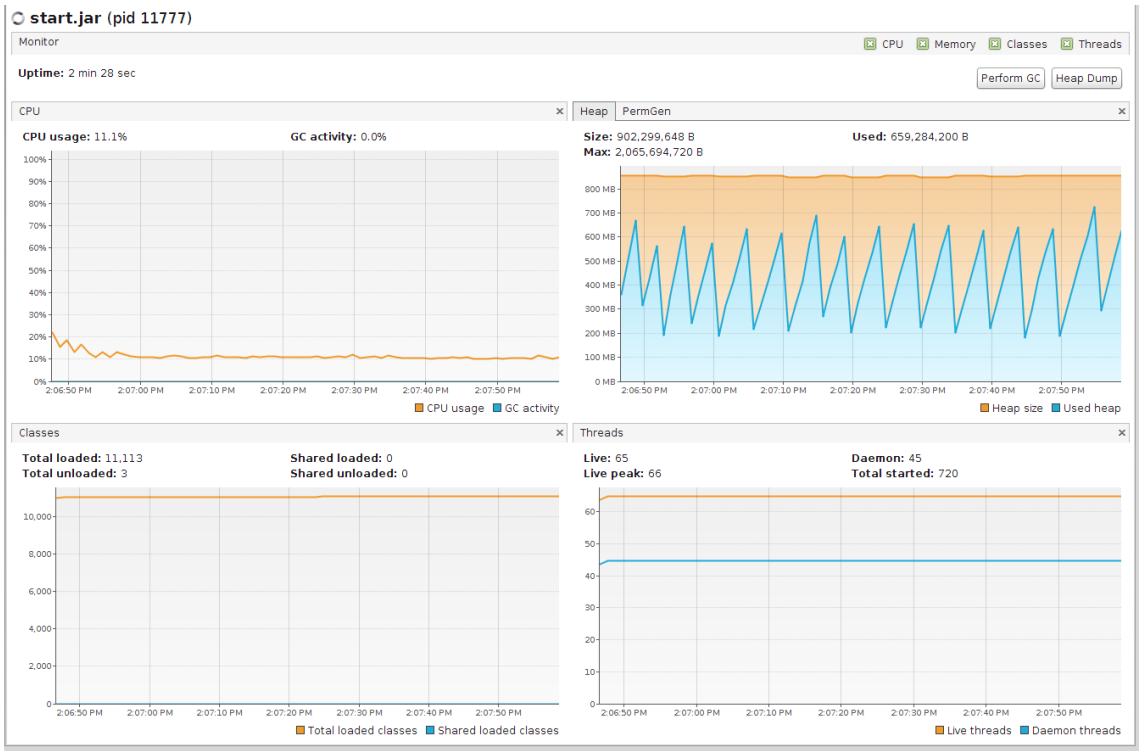

Figure 3.17: Resources used when running Exp6 with 4000 messages per minute

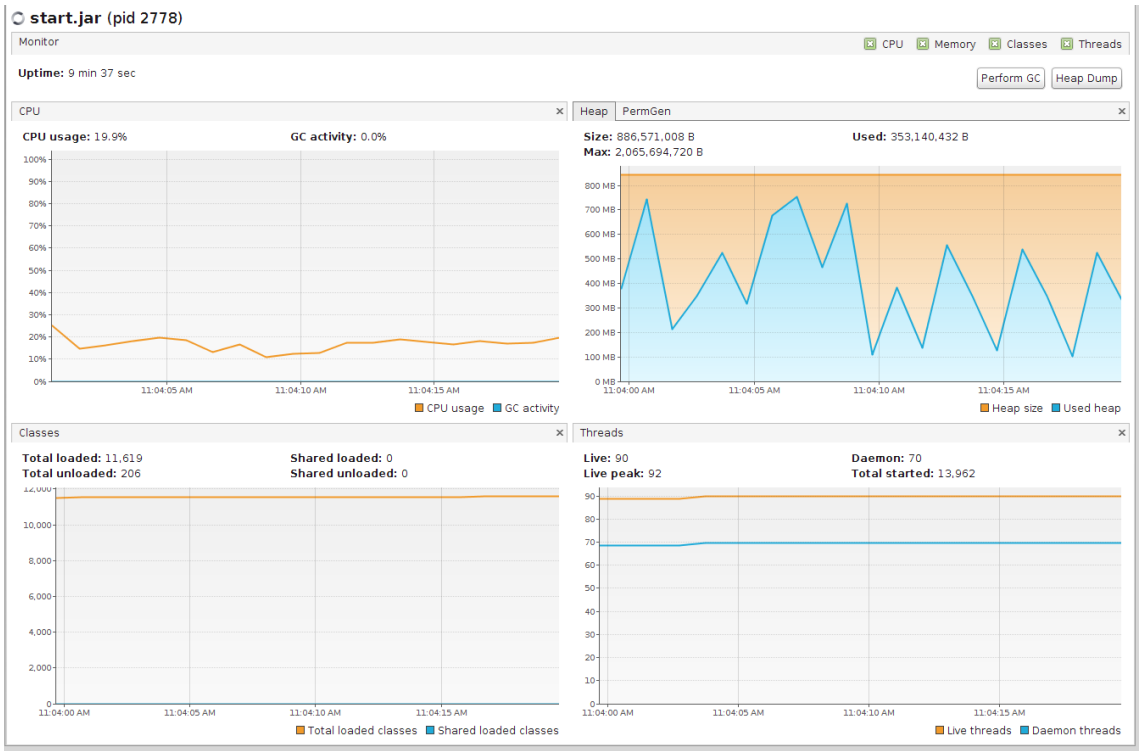

Figure 3.18: Resources used when running Exp7 with 4000 messages per minute

1 Source, 4 Sessions, 1 Client, Exp0			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	60	0	0.00
	60	0	
	60	0	
120	60	0	0.00
	60	0	
	60	0	
240	81	21	19.67
	80	20	
	78	18	
480	160	100	124.67
	191	131	
	203	143	
1000	342	282	306.67
	394	334	
	364	304	
2000	785	725	713.00
	755	695	
	779	719	
4000	1561	1501	1,480.67
	1367	1501	
	1500	1440	

1 Source, 4 Sessions, 1 Client, Exp6			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	87	27	27.33
	88	28	
	87	27	
120	89	29	29.00
	89	29	
	89	29	
240	101	41	41.00
	101	41	
	101	41	
480	162	102	101.33
	162	102	
	160	100	
1000	337	277	267.67
	335	275	
	311	251	
2000	522	462	459.67
	514	454	
	523	463	
4000	1159	1099	1,065.33
	1081	1099	
	1058	998	

1 Source, 4 Sessions, 1 Client, Exp7			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	117	57	57.67
	118	58	
	118	58	
120	119	59	59.00
	119	59	
	119	59	
240	134	74	77.67
	139	79	
	140	80	
480	221	161	146.00
	198	138	
	199	139	
1000	315	255	306.67
	441	381	
	344	284	
2000	675	615	603.67
	666	606	
	650	590	
4000	904	844	864.67
	931	871	
	939	879	

Table 3.5: 1 source, 4 sessions, 1 client middleware execution times

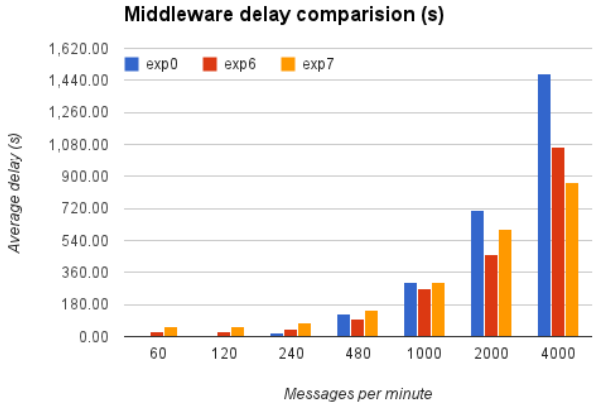

Figure 3.19: Middleware delay comparision

Additional information

Besides monitoring the CPU load, memory load and execution times, we also collected extra information related to thread execution time, CPU profiling and CPU sampling. These tests slow considerably the overall execution of the middleware and thus were conducted separately to ensure that the execution times were not compromised. Consequently, we did not run these tests for all possible scenarios because that would force us to repeat every single one of them, thus consuming a considerable amount of time without bringing any relevant data to the analysis.

The CPU profiling and sampling results gathered were all very similar, still, they provided valuable insight as to what was happening inside the middleware. Therefore, we confine the result's presentation for the heaviest case of 4000 messages per minute using Exp7 and the results we got from the Threads execution time for the same case but using both Exp0 and Exp7 Esper expressions. The remainder may be consulted in Appendix A.

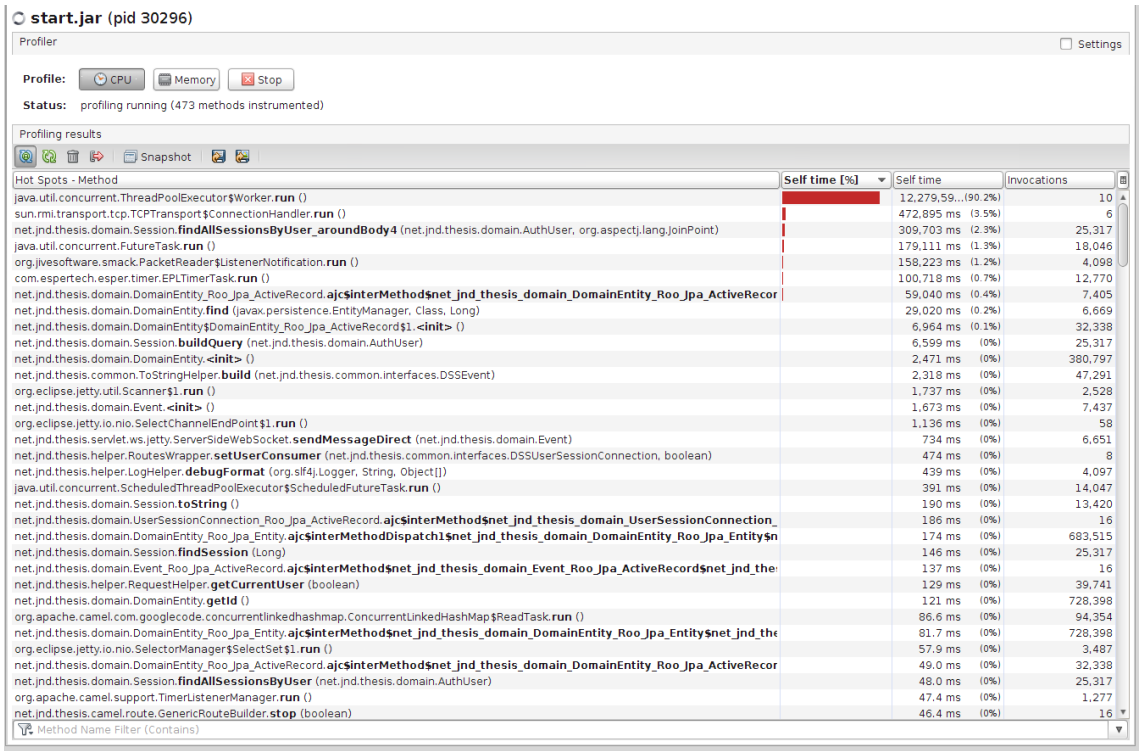

Figure 3.20: CPU profiler for 1 source, 1 session and 1 client using Exp7

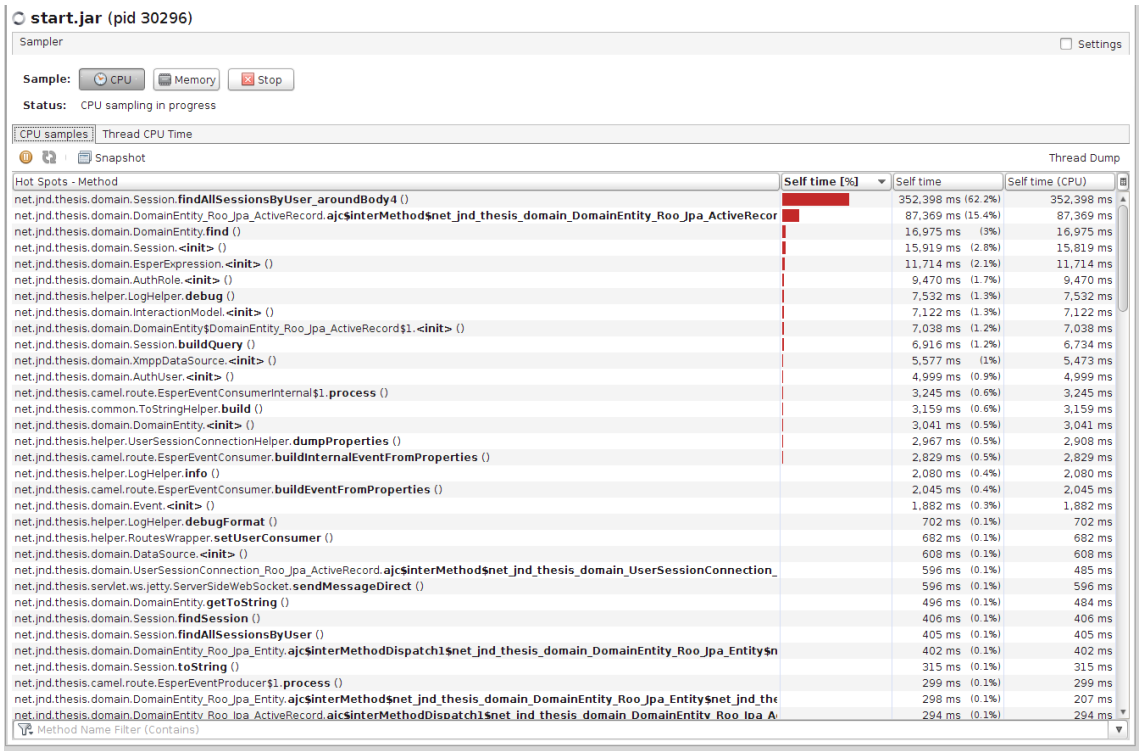

Figure 3.21: CPU samples for 1 source, 1 session and 1 client using Exp7

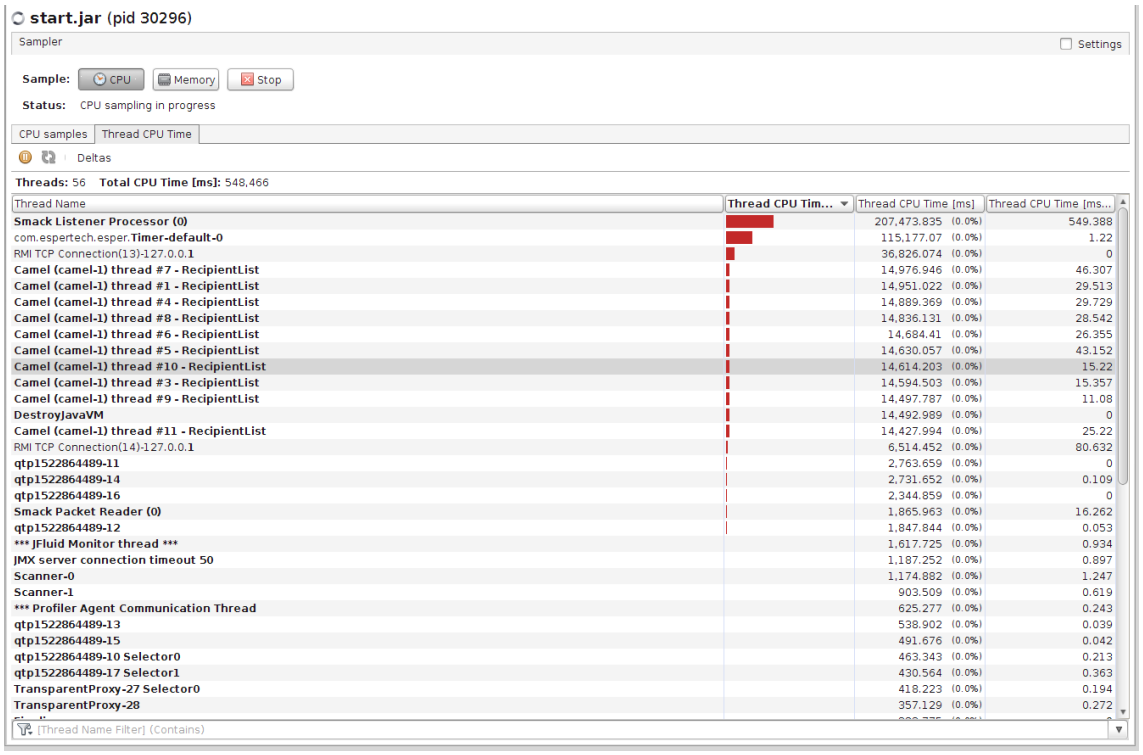

Figure 3.22: Thread time for 1 source, 1 session and 1 client using Exp7

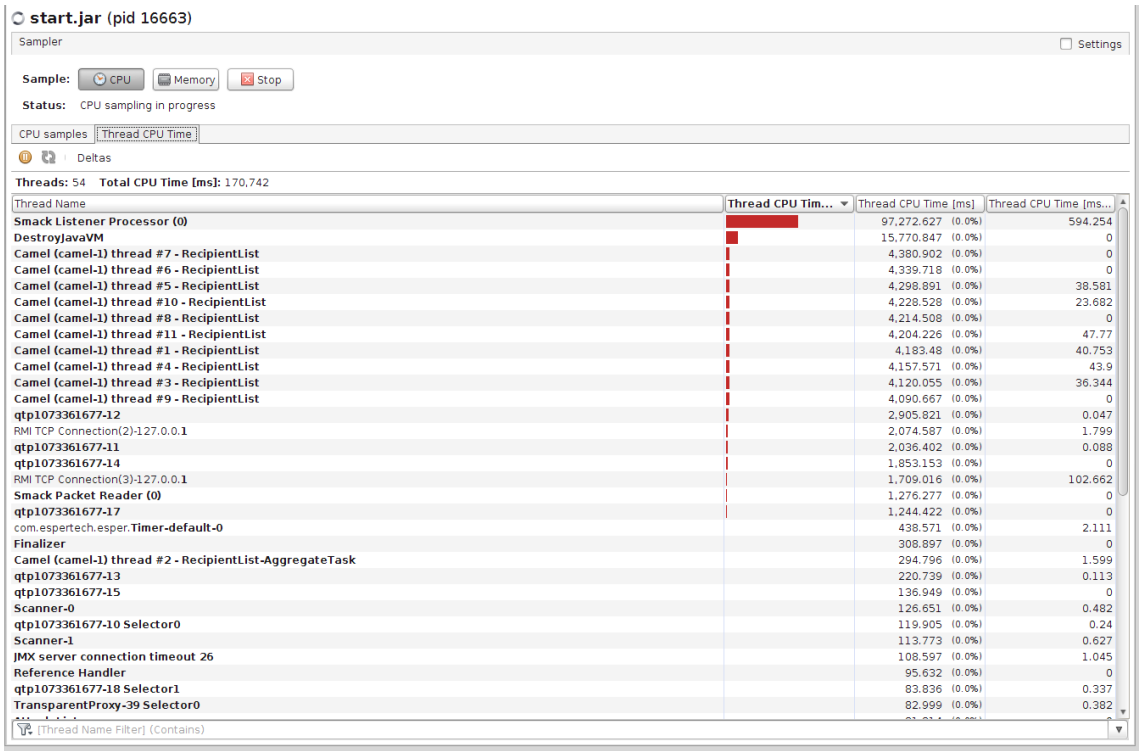

Figure 3.23: Thread time for 1 source, 1 session and 1 client using Exp0

3.2.1.5 Performance analysis

In this section we analyze the results produced by our experimental evaluation, with the purpose of identifying the performance bottlenecks.

We start our analysis by pointing out that the CPU load rarely goes above 25%. It sporadically reaches values between 25% and 30% but those are only reached when the entire infrastructure: middleware, Jetty web server, etc. are being launched. Once the load stabilizes, it remains near 20%, which indicates that the middleware is only using one of the eight hardware threads that the underlying CPU features. This remains true for all scenarios, independently of how many sessions are running, of the Esper expression being used, and of how many messages the data-source sends per minute, as can be observed in the graphs and screenshots from section 3.2.1.4, and can be further confirmed in sections A.1, A.2 and A.3.

As for the delay between the source's execution time and the middleware's execution time, one can observe that it is significant. In the first scenario with only one session and using the simplest Esper expression, this delay grows up to 512.33 seconds, which is roughly 8.53 minutes. Having in mind that our source only executed for sixty seconds, that there is only one session and one client running, and that no processing at all is being done, this is a considerable delay with a big impact in the system's performance. For this simple case, as the number of sessions grow and the Esper expression performs more computations over time, the delay gets worse as expected.

A closer look however, will point an inconsistency in the graphs. Even with the simplest scenario evaluated in the last paragraph, there is one battery of tests that does not behave as expected - the test where the data-source is sending 2000 messages per minute. In this case, the Esper expression calculating the average of the last thirty seconds (Exp6) is faster than the Esper expression that lets everything pass (Exp0). This inconsistency is even more visible when running the scenarios with two and four active sessions at the same time.

The results from Figures 3.24 and 3.25 do not reveal much. The result obtained from the sampler is to broad to be of any use, and the result from the profiler also does not pinpoint anything surprising - it just tells us that there are a lot of threads executing, but we do not know where or who created them. Finally, figures 3.26 and 3.27 confirm something that makes sense, when using Exp0 our CPU wastes no time using the Esper engine, but when we use the Esper expression Exp7, the result differs and the CPU threads actually spend some time inside the Esper engine.

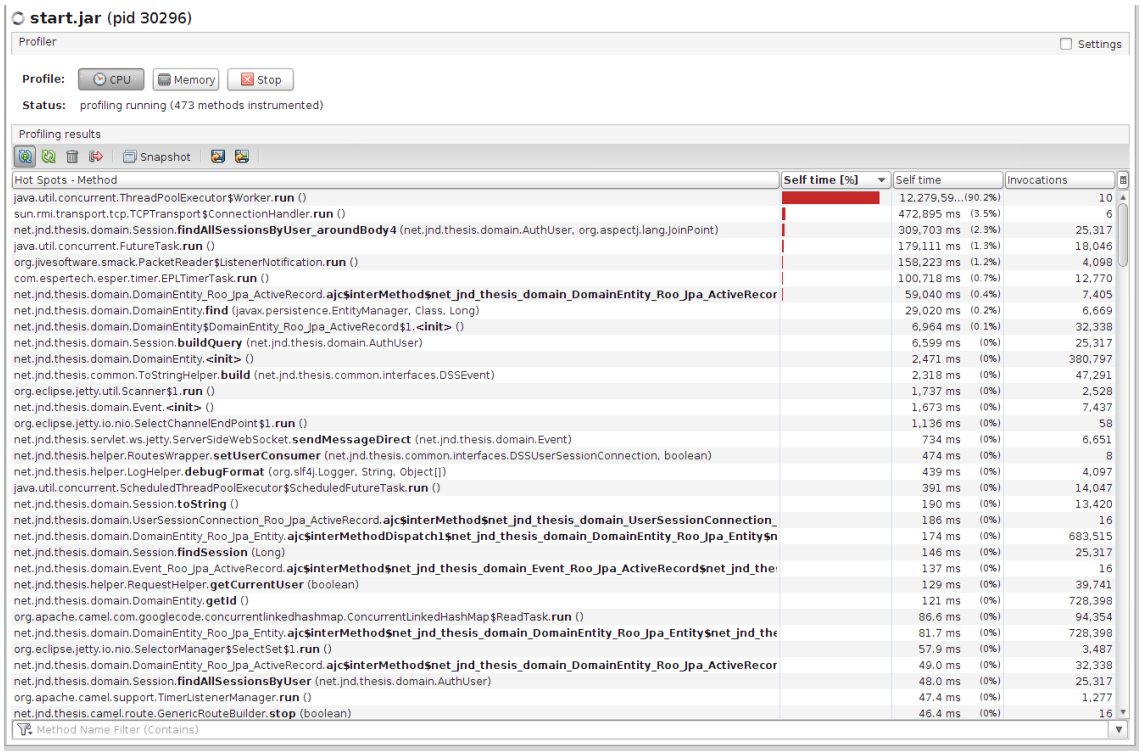

Figure 3.24: CPU profiler for the case of 4000 messages per minute using Exp7

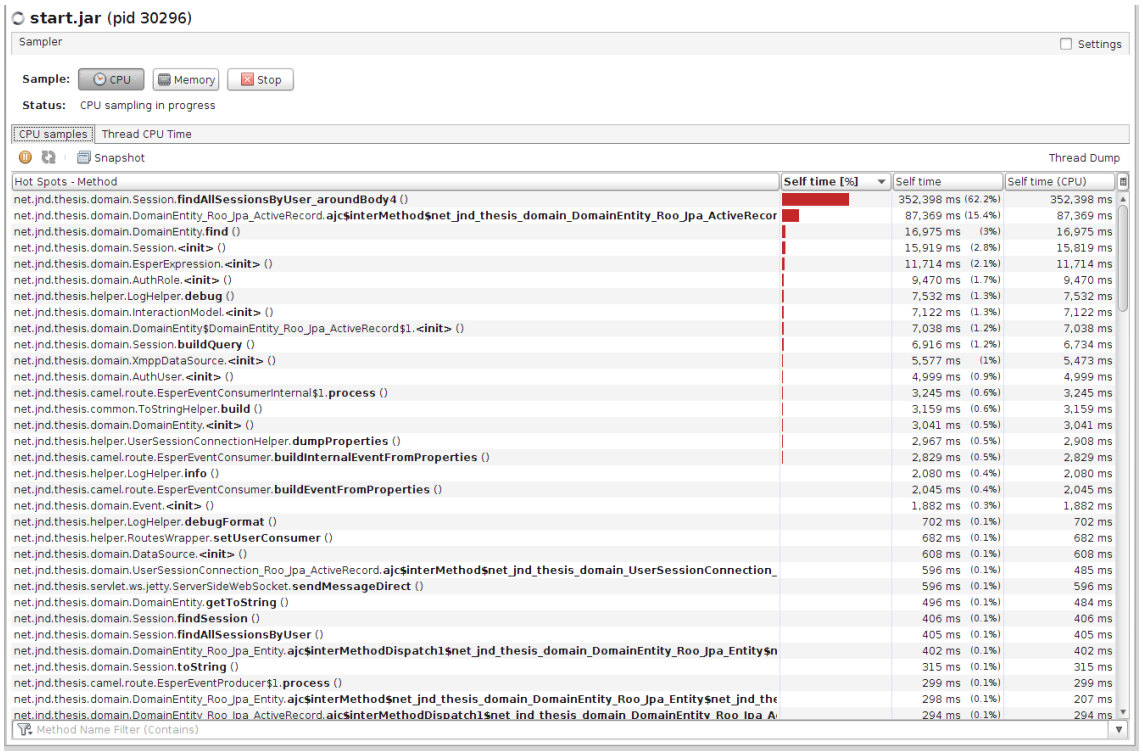

Figure 3.25: CPU sampler for the case of 4000 messages per minute using Exp7

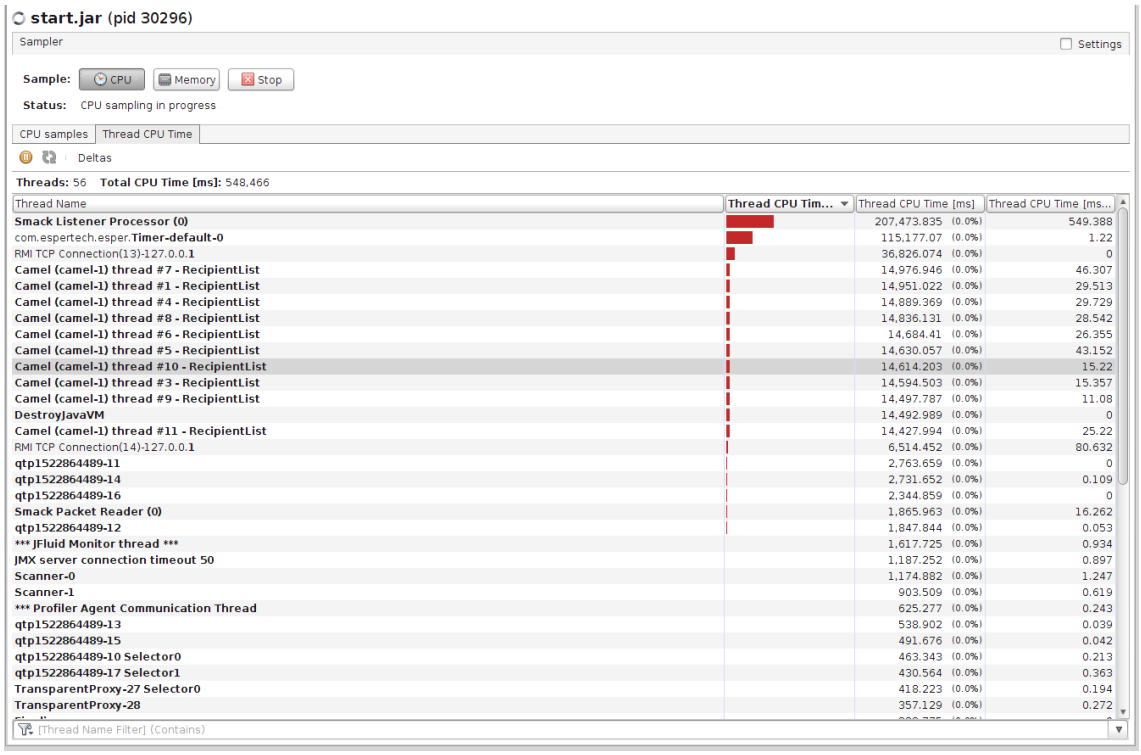

Figure 3.26: CPU threads for the case of 4000 messages per minute using Exp7

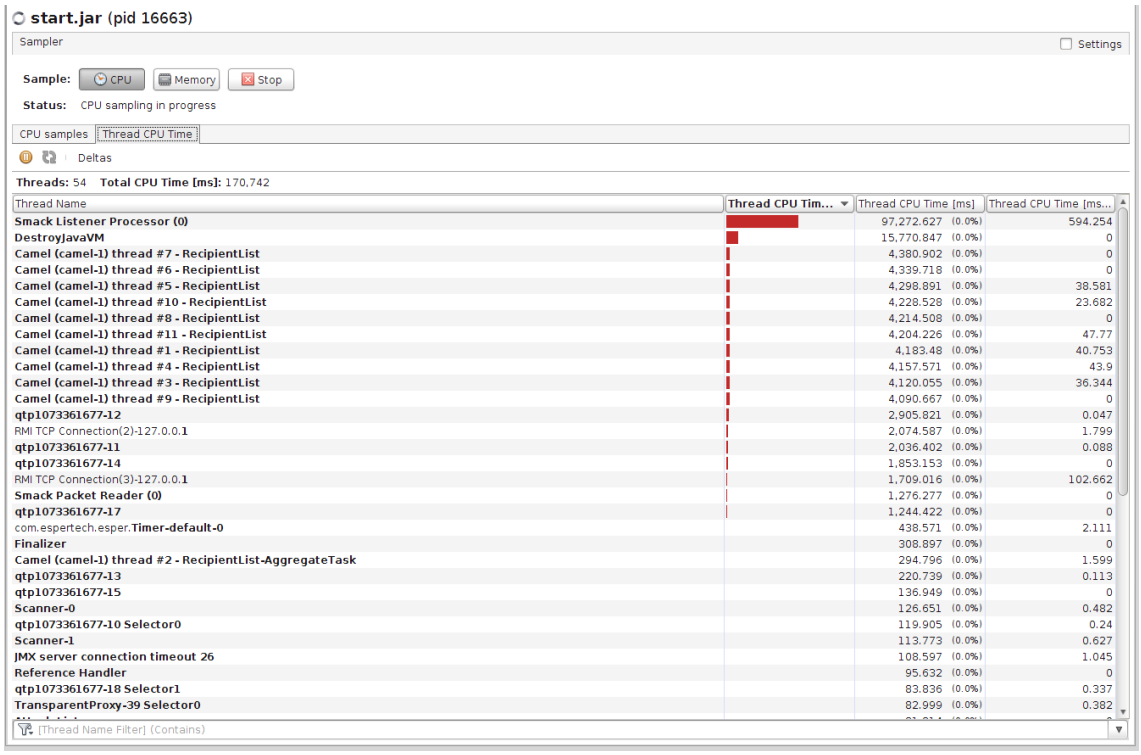

Figure 3.27: CPU threads for the case of 4000 messages per minute using Exp0

This first set of experiments confirmed that the middleware's performance is far from being perfect, now the question that we ask is, *What could be causing these performance problems?*. Given the nature of the tests we have reason to believe that the root of the problem can either be two areas:

- Inside the data-source layer of the middleware component, where the messages are first received and treated by the camel routes;
- Inside the session messaging layer where the messages are treated by the Esper endpoints.

To find the source of the problem, a more profound analysis had to be made - an operational analysis.

3.2.1.6 Operational analysis

To really understand the root of evil, we had to go deep into the middleware. This involved discovering and analyzing what was using more resources inside the middleware, and how we could improve it.

By analyzing the graphs provided by jvisualVM we concluded that the problem could either be in the Esper endpoints, processing the messages, or in the Camel routes connecting them.

Could Esper processing be the problem?

When the middleware is not using the Esper engine, for example when using expression Exp0, the results are as expected: the average delay increases proportionally to the increase of messages sent by the data-source.

However, when we use the Esper engine to calculate averages (like in expressions Exp6 and Exp7) or to perform other computations, Esper uses all the enhancements it knows to make the process as fast as possible. This means that even though we have more messages and a bigger work load, because Esper is now doing work on its own, the average delays will be smaller. This is specially visible in the scenarios where we tested we two sessions and four sessions - here the difference in the delays from Exp0 to Exp6 and Exp7 is very high. In fact, the delay from Exp0 to Exp7 in the test with four sessions is nearly reduced by half.

Consequently, having in mind that Esper is already very efficient at calculating averages, that the proposed solution would be highly complex, and the the impact of this improvement in the middleware would be minimal, we decided to not implement it and move on the other direction. This solution however is still viable, and it would be feasible to implement it in the future in a long run.

Could Camel routes be the bottleneck?

With the Esper option discarded we were left with the Camel routes option. This theory was based in the formulation that the camel routes were single threaded and thus were not processing and delivering the data as fast as they should.

However, after following the same modus operandi of making a research through the documentation and asking the Camel community for help no conclusions were drawn. Camel is complex and it behaves differently depending on specific situations so, not only is this the documentation scarce, but also no one really knows how it works as well. Thus, in order to draw our own conclusions in the scope of the middleware under evaluation, we had to conduct specific tests to discover how many threads Camel was using, and which parts of the system they were running.

To implement these tests we had to make a few changes in the middleware, so it could output which threads were being used and when. Then, after making the middleware run for a minute with one of the previous tests, we would then check the log files by human eye and draw the necessary conclusions.

Therefore, after running the middleware several times and studying the logs, we concluded that the camel routes inside the middleware core are affected mainly by three parameters:

- The number and type of data-sources sending information
- The number of sessions currently active in the middleware
- The number of clients connected to each session

These three parameters affect the number of threads being used by the core, and therefore directly affect the rate of messages that the core can handle at any given time.

Basic scenario - 1 data-source, 1 session, 1 client

In the simplest of scenarios the three layers of the Core component are executed by only two threads, thread A and thread B, as seen in the logs:

Listing 3.4: Thread A carrying information from the data-source XMPP endpoint to the first Esper endpoint

```
1 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer, process(),
     from route: xmpp://localhost:5222/?user=thesis.dimfccs@gmail.com&password
     =thesis.dimfccs.pwd&nickname=Source1
2 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer, process(),
     to route: esper://session_1
3 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer,  process(),
     threadId: 47
```

Listing 3.5: Thread A carrying information from the first Esper endpoint to the second one

```
1 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
      EsperEventConsumerInternal, process(), from route: esper://session_1?eql=
      select * from pattern [every e=net.jnd.thesis.camel.bean.
      CamelInternalEvent(sid=0 and sid<1394391793074)]
2 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
      EsperEventConsumerInternal, process(), to route: esper://session_1
3 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
      EsperEventConsumerInternal, process(), threadId: 47
```

Listing 3.6: Thread B carrying information from the second Esper endpoint server websockets

```
1 [DEBUG][route.EsperEventConsumerUser]: DEBUG_PATH EsperEventConsumerUser,
      process(), from route: esper://session_1_1?eql=select * from pattern [
      every e=net.jnd.thesis.domain.Event(session.id=1 and session.id
      <1394391793091)]
2 [DEBUG][route.EsperEventConsumerUser]: DEBUG_PATH EsperEventConsumerUser,
      process(), threadId: 49
3 [DEBUG][jetty.ServerSideWebSocket]: ServerSideWebsocket, sendMessageDirect:
      time in Middleware: 224
```

Thread A is responsible for the processing of all the data that arrives, from the data-source messaging layer to the session messaging layers.

Thread B is responsible for the client messaging layer - it picks information from E2, processes it through R3 and then sends it to the client using websockets. Thread B will not process any data until thread A finishes, meaning that data processment is sequential from thread A to B.

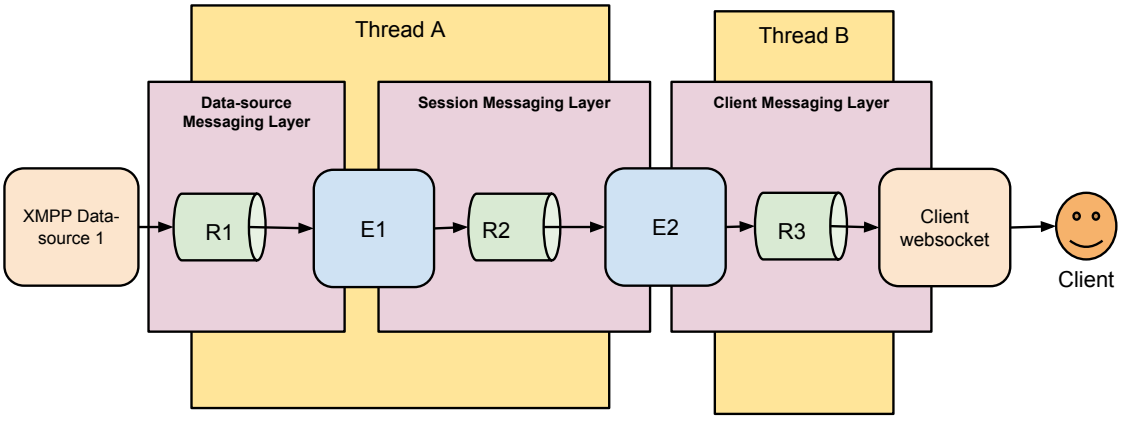

Figure 3.28: Threads used by the middleware core in the 1 source, 1 session and 1 client scenario

Multiple Sessions - 1 data-source, 2 session, 1 client

When there are multiple active sessions, the middleware creates distinct Esper endpoints for each session in the Session Messaging Layer.

This way, if R1 receives a message, it delivers it to the respective Esper endpoints (E1 S1 and E1 S2 in Figure 3.29) associated to the multiple sessions reading from the data-source in study. The messages are then processed in the scope of each session, following its routing scheme, and forwarded to the final Esper endpoint. In the end, R3 retrieves the messages from the final Esper endpoint associated to the client's session and sends them to him through websockets.

During this scenario only two threads are ever used as well - threads A and B. This holds true independently of the number of sessions or clients that exist. This means that if there are too many sessions, thread A will become a bottleneck to performance and it will slow down the system because it has to deal with the request sequentially before passing it to thread B. Because the number of threads never scales, as a consequence, the CPU usage level never goes above 30%, meaning that we do not take full advantage of the machine's multi-core architecture.

Listing 3.7: Thread A carrying information from the data-source XMPP endpoint to the first Esper endpoint

```
1 [DEBUG][route.EsperEventProducer]: DEBUG_PATH EsperEventProducer, process(),
    from route: xmpp://localhost:5222/?user=thesis.dimfccs@gmail.com&password
    =thesis.dimfccs.pwd&nickname=Source1
2 [DEBUG][route.EsperEventProducer]: DEBUG_PATH EsperEventProducer, process(),
     to route: esper://session_2
3 [DEBUG][route.EsperEventProducer]: DEBUG_PATH EsperEventProducer, process(),
    threadId: 55
```

Listing 3.8: Thread A carrying information from the first Esper endpoint to the second one

```
1 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
    EsperEventConsumerInternal, process(), from route: esper://session_1?eql=
    select * from pattern [every e=net.jnd.thesis.camel.bean.
    CamelInternalEvent(sid=0 and sid<1394891720996)]
2 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
    EsperEventConsumerInternal, process(), to route: esper://session_1
3 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
    EsperEventConsumerInternal, process(), threadId: 55
```

Listing 3.9: Thread B carrying information from the second Esper endpoint server websockets

```
1  [DEBUG][route.EsperEventConsumerUser]: DEBUG_PATH EsperEventConsumerUser,
       process(), from route: esper://session_1_1?eql=select * from pattern [
       every e=net.jnd.thesis.domain.Event(session.id=1 and session.id
       <1394891721006)]
2  [DEBUG][route.EsperEventConsumerUser]: DEBUG_PATH EsperEventConsumerUser,
       process(), threadId: 57
3  [DEBUG][jetty.ServerSideWebSocket]: ServerSideWebsocket, sendMessageDirect:
       time in Middleware: 162
```

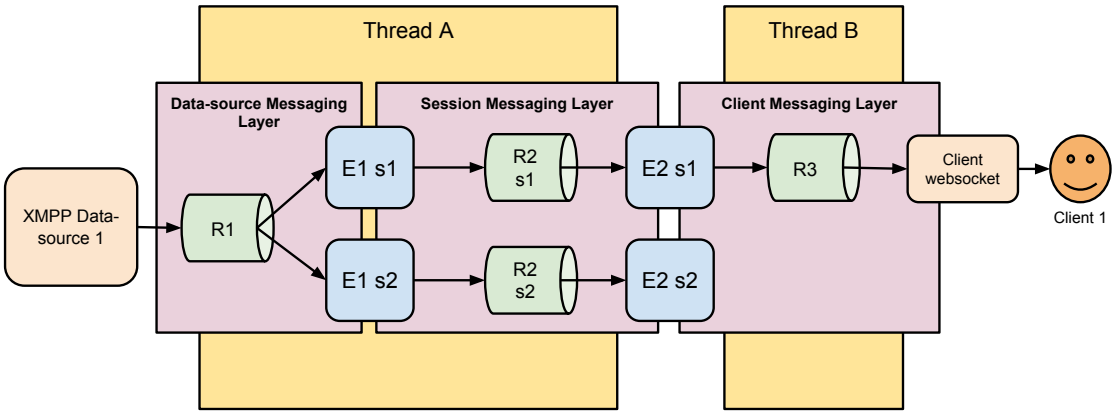

Figure 3.29: Threads used by the middleware core in the 1 source, multiple sessions and 1 client scenario

Multiple data-sources - 2 data-sources, 1 session, 1 client

With multiple data-sources, the middleware is more flexible as it creates two threads for each source. Thus, if there are two XMPP data-sources there will be four threads: A, B, C and D.

Threads A and B will be used to process the information from the first data-source, while threads C and D will process information from the second data-source. Each thread takes care of the same actions that it would if there was only a single source, like in Figure 3.28.

Both logs and Figure 3.30 illustrate this behavior:

Listing 3.10: Thread A carrying information from the first data-source XMPP endpoint to the first Esper endpoint

```
1 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer, process(),
    from route: xmpp://localhost:5222/?user=thesis.dimfccs@gmail.com&password
    =thesis.dimfccs.pwd&nickname=Source1
2 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer, process(),
    to route: esper://session_1
3 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer,  process(),
    threadId: 47
```

Listing 3.11: Thread C carrying information from the second data-source XMPP endpoint to the first Esper endpoint

```
1 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer, process(),
    from route: xmpp://localhost:5222/?user=thesis.secondary.source@gmail.com
    &password=thesis.secondary.source&nickname=Source2
2 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer, process(),
    to route: esper://session_1
3 [DEBUG][route.XmppEventProducer]: DEBUG_PATH XmppEventProducer,  process(),
    threadId: 52
```

Listing 3.12: Thread A carrying information from the first Esper endpoint to the second one

```
1 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
    EsperEventConsumerInternal, process(), from route: esper://session_1?eql=
    select * from pattern [every e=net.jnd.thesis.camel.bean.
    CamelInternalEvent(sid=0 and sid<1393109157335)]
2 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
    EsperEventConsumerInternal, process(), to route: esper://session_1
3 [DEBUG][route.EsperEventConsumerInternal]: DEBUG_PATH
    EsperEventConsumerInternal, process(), threadId: 47
```

Listing 3.13: Thread B carrying information from the second Esper endpoint server websockets

```
1 [DEBUG][route.EsperEventConsumerUser]: DEBUG_PATH EsperEventConsumerUser,
    process(), from route: esper://session_1_1?eql=select * from pattern [
    every e=net.jnd.thesis.domain.Event(session.id=1 and session.id
    <1393109157349)]
2 [DEBUG][route.EsperEventConsumerUser]: DEBUG_PATH EsperEventConsumerUser,
    process(), threadId: 54
3 [DEBUG][jetty.ServerSideWebSocket]: ServerSideWebsocket, sendMessageDirect:
    time in Middleware: 190
```

Here thread A is responsible for the data-source and session messaging layers of source 1, and thread B is responsible for the client messaging layer. Threads C and D take up the same duties as threads A and B, but instead of doing it for the first data-source, they do it for second. This behavior holds true even if both data-sources have different types.

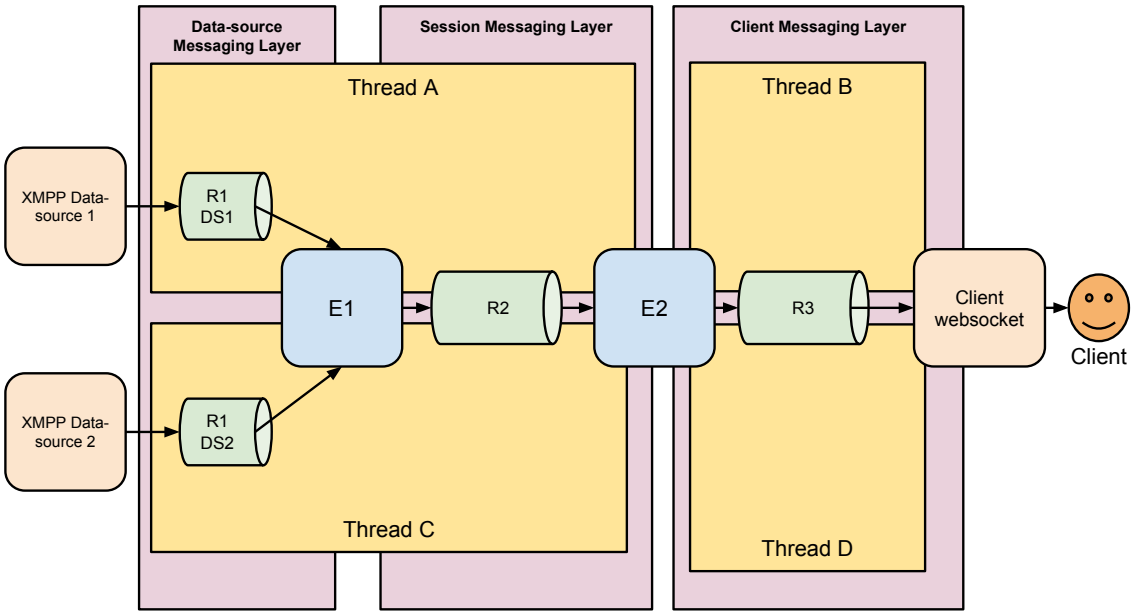

Figure 3.30: Threads used by the middleware core in a scenario with multiple sources, 1 sessions and 1 client

Multiple clients - 1 data-source, 1 session, 2 clients

The middleware is prepared to distribute information to multiple clients. During this scenario, thread B in the client messaging layer will forward the messages to each of the client's websockets in a round-robin manner by using a for loop, as described in the figure below.

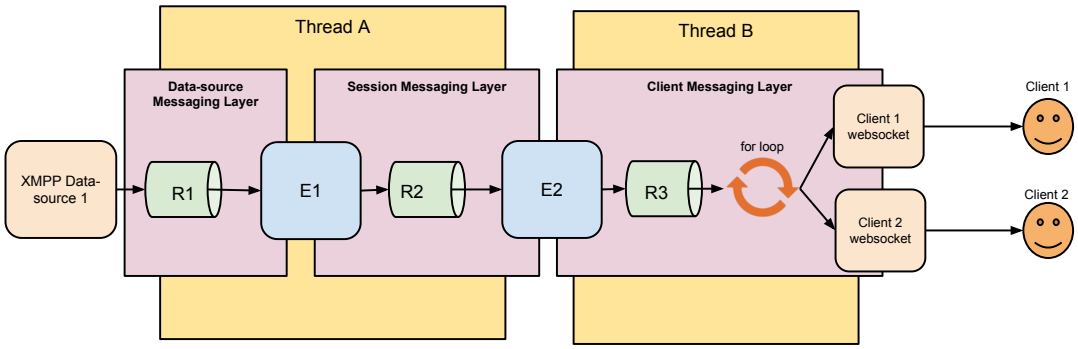

Figure 3.31: Threads used by the middleware core in a scenario with 1 data-source, 1 session and 2 clients

Discussion

During the evaluation of the architecture two main faults were found, one in the scenario with multiple sessions and the other in the scenario with multiple clients.

In the first scenario, depicted by Figure 3.29, we concluded that the middleware does not indeed take advantage of multicore architectures - as one can see, thread A does the majority of the work. In an application where there is expected to be tenths of or even hundreds of sessions listening to data from the same data-source this clearly becomes a problem for thread A, which has to deal with the processing inherent to all of them. Therefore, this results in messages pilling up and waiting to be processed, which slows the overall execution flow and works as a bottleneck.

In the second scenario, depicted by Figure 3.31, the processed messages are delivered to the clients in a round-robin fashion. This means that thread B sends a message to one client, then sends the exact same message to another client, and so on, one by one. This is time consuming because with hundreds of clients, the last client to get the message suffers a considerable delay.

3.2.2 Data Layer

Having in mind the nature of the middleware, scalability is of the greatest importance. So far we have addressed the scalability problems by trying to study the performance of the middleware and trying to improve it so it can handle more requests and therefore scale better. However, there is yet another level one can study to improve the middleware's scalability - the data layer.

All the data that goes through the middleware is saved in a MySQL database, a local repository to the machine that allows it to revisit old sessions should there be a need for such. However, MySQL is a Relational Database Management System (RDBMS), and these systems have several problems when it comes to scalability and real-time data handling [mon; Har], such as:

- These systems do not scale easily. They usually scale up (buy a bigger server as load increases) instead of scaling out (distributing the database across multiple machines as load increases).
- Although RDBMS can use sharding, doing so usually implies loosing certain benefits and involves a complex adaptation process. Furthermore the machines used for this process have to be reliable and are usually not cheap.
- The project involves Big Data management, and RDBMS struggle with such huge volumes of data when compared to other solutions.
- No private clouds currently support a scalable approach to RDBMS, nor are they compatible with the RDBMS AWS service that the middleware uses.

Furthermore, because mapping relational objects to POJO's is not trivial [IBNW09], the middleware has to use an extra layer to do that work, which can have a strong impact in the performance of the system once it hits high workloads.

4

Proposed Optimizations

In this chapter we propose a set of solutions for the problems found in section 3.2.1.

Hence, section 4.1 starts with proposals of solutions for the performance problems found sections 3.2.1.5 and 3.2.1.6, and section 4.2 presents alternatives to the data layer and how changing it may improve the middleware's performance and scalability.

4.1 Performance

In Chapter 3 we analyzed the performance of the middleware regarding a pre-defined set of parameters, and subsequently identified several performance issues. In this section we propose solutions to those problems and implement a subset of these proposals. To assess their impact we analyze their results and evaluate if we achieved our goals of increasing performance.

4.1.1 Proposals

In Section 4.1 we presented two distinct scenarios that illustrate the current problems of the middleware:

- In the first scenario, depicted by figure 3.29, we concluded that one of the threads was being overwhelmed with work.
- In the second scenario, depicted by figure 3.31, we realized that the delivery of messages to the clients was not being as efficient as possible.

In this section we attack both scenarios separately and try to find solutions for the problems that they represent.

4.1.1.1 Proposed solutions for scenario 1

To fix this problem we propose the parallelization of the Camel route in the data-source messaging layer. This parallelization can be done at three points:

1. At the entrance of the Camel route, so it can receive more data from the Openfire server without suffering a bottleneck;
2. During the processment of the messages in the Camel route;
3. In the end of the Camel route where the delivery to the Esper endpoints is done.

Fortunately, Camel has a good array of parallelization options. The following options are available:

- Using several Enterprise Integration Patterns (EIP)[1] patterns that support concurrency models
- Using special components for that effect, like SEDA[2]
- Using the Threads DSL[3]
- Using ServicePool[4] for pooling services
- Making use of components that already make use of pooling, such as JMS[5]

However, not all these solutions however can be used.

The EIP list does not have any patterns that could be easily used. The pattern that embeds a behavior closer to our goal is the Recipient List pattern, but this one requires a static set of endpoints, which simply does not happen because sessions are created and destroyed dynamically throughout the middleware's execution. In order to introduce dynamic routing in the Recipient List pattern, we would have to enclose some sort of indication on in the messages about where they should go. However, such approach is currently not possible, given that the messages that flow on the middleware do not have a specific destination. The only possible way to make this approach work, would be to remake sections of the middleware into having more information about the routes and sessions. However, given the fact this solution goes beyond the objectives of this work - the optimization of the current implementation - into the territory of reimplementation, we opted to discard it.

The ServicePool class is merely for the specific case of the Camel producer service and it is restricted to pooling. Therefore it makes no sense to consider it for this scenario.

Finally, the JMS component, which is a rival of SEDA, is considerably less efficient because it reuses Spring 2's JmsTemplate for sending messages. Furthermore, because

[1] https://camel.apache.org/eip.html
[2] http://www.eecs.harvard.edu/~mdw/proj/seda/
[3] https://camel.apache.org/async.html
[4] https://camel.apache.org/servicepool.html
[5] https://camel.apache.org/jms.html

it has more features that go outside the parallelization inside the same JVM, fail-over support and machine clustering, it takes extra setup and configuration.

Passed this initial sieve, we were left with the Threads DSL and the SEDA component options.

The use of the Threads DSL consists mainly in using the Java DSL to define thread pools at specific points in the route. To improve this solution, one can use an Executor[6] to have additional control over the pool. In theory, having thread pools when receiving data and thread pools when processing it would significantly reduce the load charge on thread A, and consequently improve efficiency. However, because we cannot predict the speed at which different threads process different messages, the order of those messages would be lost.

The second option, using the SEDA component[7], is a little more complex. SEDA stands for Staged Event Driven Architecture, and it allows for the decomposition of complex event-driven applications into sets of stages connected by queues. DSL also employs dynamic control to automatically tune runtime parameters and it also capable of managing load.

In Camel the DSL component provides asynchronous behavior, more specifically, it allows for consumers to be invoked in separate threads from the producer. This means that instead of having only one thread processing and delivering the data to all the Esper endpoints, we would have a thread receiving and processing the data, and multiple threads receiving it in the respective Esper endpoints.

Thus, we this knowledge, we propose using these two parallelization techniques, the Java threads DSL and DSL, at the various stages of the Camel route.

4.1.1.2 Proposed solutions for scenario 2

Remembering Figure 3.31, we concluded that the delivery of messages to the clients was not being as efficient as possible. To fix this problem, and since the content of the message is the same for a subset of clients, one could try to multicast the message to all of them at the same time. Our first attempt to fix the problem was to consider using the Camel Recipient List[8] EIP with the *parallelProcessing* flag set to true. However, this solution is not compatible with the Websocket technology used in the middleware's client-directed communication, and so we were forced to discard it.

Another possible solution would be to parallelize the emission of the messages, by using Java threads. However, having in mind that we want to keep the order of the messages and that the message being multicast to the clients is the same, this solution would pose several challenges.

First, the queue where the messages are stored only has one copy of each message. So having multiple threads consuming from that queue would not guarantee the multicast of

[6] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html
[7] https://camel.apache.org/seda.html
[8] http://camel.apache.org/recipient-list.html

that one message to all the clients. For this to work, the thread that first pulls the message out of the queue, would have to somehow communicate with all the other threads and give them a copy of the message or tell them who has the message so the other threads could later on pick it up. Another solution to this problem would be to change the queue to only remove the message once all threads had a copy of it. This however would force the queue to know how many threads are using it, which could pose a problem since the number of threads using a queue may vary over time, and the queue would be dependent on that number.

Second, we have the order problem. Because each thread consuming from the queue can send messages to multiple clients, and each client can receive messages from multiple threads, some will process information faster than others and order cannot be guaranteed. Furthermore, it is also not possible to prevent a client from receiving the same message multiple times from different threads, because by just looking at the messages in the queue it is impossible to tell which clients have already received them. To avoid this one could make the process of removing a message from the queue atomic, by using a lock mechanism to force the other threads to wait, or one could add a time stamp or an order identifier to a message. For the first solution, one would have to consider pessimistic concurrency control (lock-based) instead of optimistic (transactional) because the chance for conflict and therefore rollback would be considerably elevated. For the second solution, the client would have to buffer all the received messages and then order them by according to the time stamp or order identifier.

Given these challenges and to alleviate the problems they represent, one could have a thread per group of clients or even a thread per each session. To have a thread per group of clients one would have to first define what a group of clients is, how big should it be and the characteristics of the clients inside that said group. A possible setup would be to have groups with clients that share the same filtering expressions in their end. Having a thread per session would ease the problems with messages, however it would be significantly less effective than the other more fine grained solutions. Even then, with this propositions, one would still have to attack the previously mentioned challenges in one way or another.

A last possible solution for this problem would be to use the new Java Non-blocking I/O (NIO) API[9]. This new API allows a single thread to take care of requests asynchronously using a low level non-blocking API that is based in channels, buffers and selectors. By using this technology, one could have a thread reading data from a buffer containing all the messages to be sent to a client, and then redirect that data to client specific buffers via channels. The problem with this approach is that the full-duplex communication behavior of Websockets would have to be re-implemented by hand again. Furthermore this alternative would also require considerable changes not just to the client messaging layer of the middleware, but also to the client applications created to work specifically with the already existing architecture.

[9] http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html

4.1.2 Implemented solutions

Our implementation work focused on the data-source and session messaging layers, where the middleware's performance is more critical and raises more issues. Of the two proposals we restricted ourselves to the first one, elaborated in section 4.1.1.1, using DSL, which will relieve Thread A of some workload.

Here we explain how this implementation affects the overall architecture and in the end we show the performance gains yield by this optimization.

4.1.2.1 SEDA

As previously seen in Section 4.1.1.1, this solution allows for a producer and multiple concurrent consumers to run on separate threads. To enable such decoupling, due to limitations of the current software architecture, an additional endpoint had to be created in order to fully take advantage of DSL (see DSL endpoint in Figure 4.1). This new endpoint receives the data right after thread A is done with the post-processing, and then sends the messages to all the different Esper queues depending on the sessions. It is important to notice that in this solution the processment of data is parallelized thanks to the DSL component, unlike the processment of data done in 3.29.

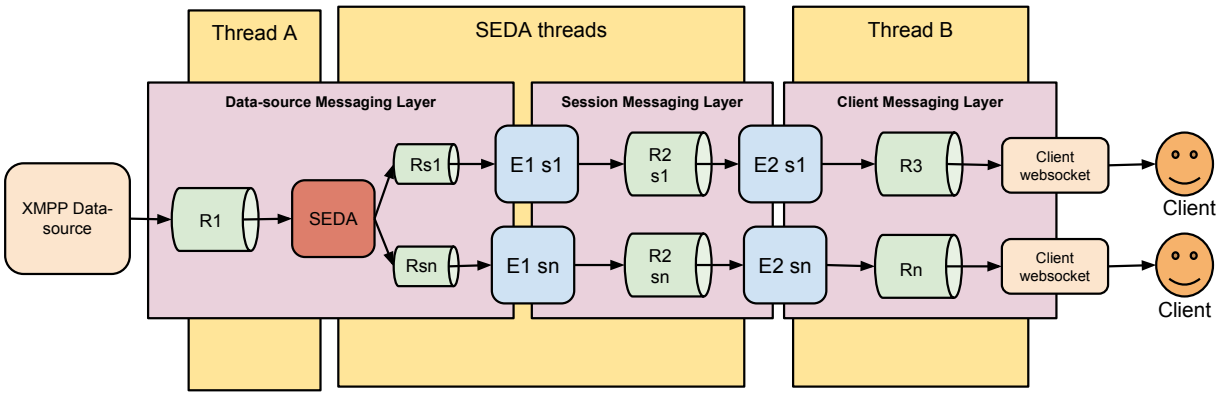

Figure 4.1: SEDA implementation

The DSL solution however, does have some drawbacks. First we are not sure if the order of messages is preserved. Second, should a critical failure occur, like a JVM crash, if the information was not already saved in the database, then that information will be lost because DSL is not prepared for it. However, since this is an unlikely scenario that is

out of the scope of the thesis, we are not concerned with it, neither did we evaluate its implications.

4.1.3 Experimental results

In this section we present the results obtained with the DSL version of the middleware. Then we move on to a performance analysis where we evaluate the outcome and take conclusions.

4.1.3.1 SEDA results

1 source, 1 session, 1 client, SEDA

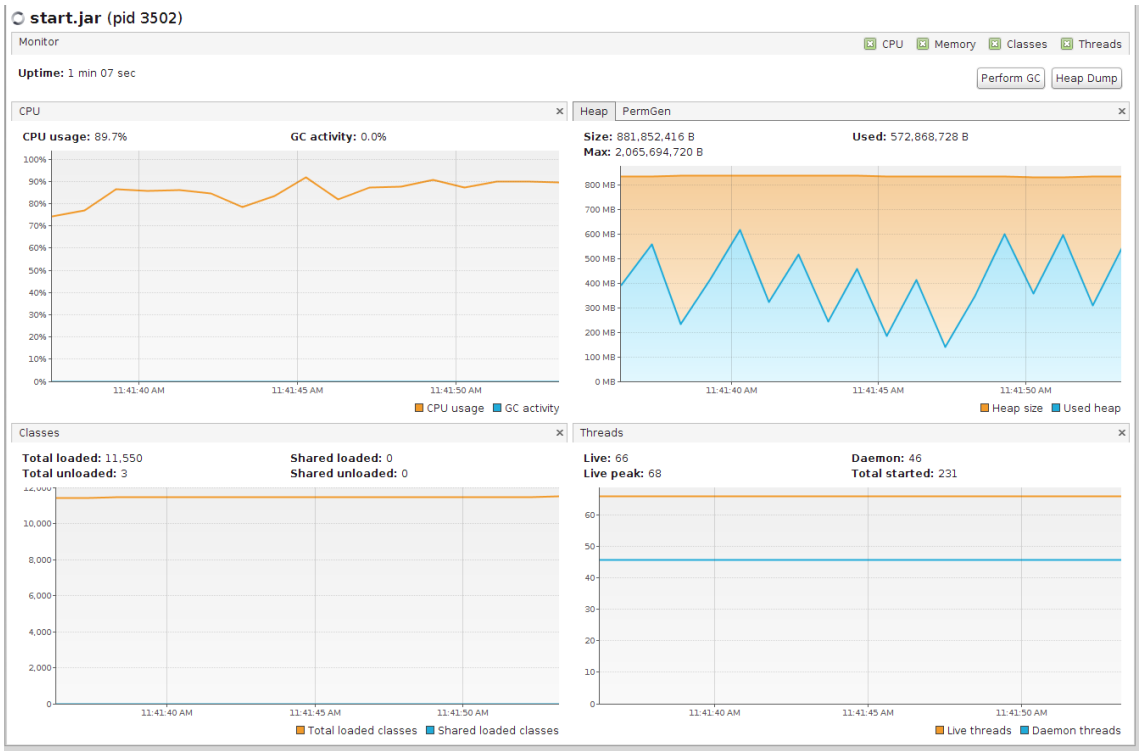

Figure 4.2: Resources used when running Exp0 with 4000 messages per minute

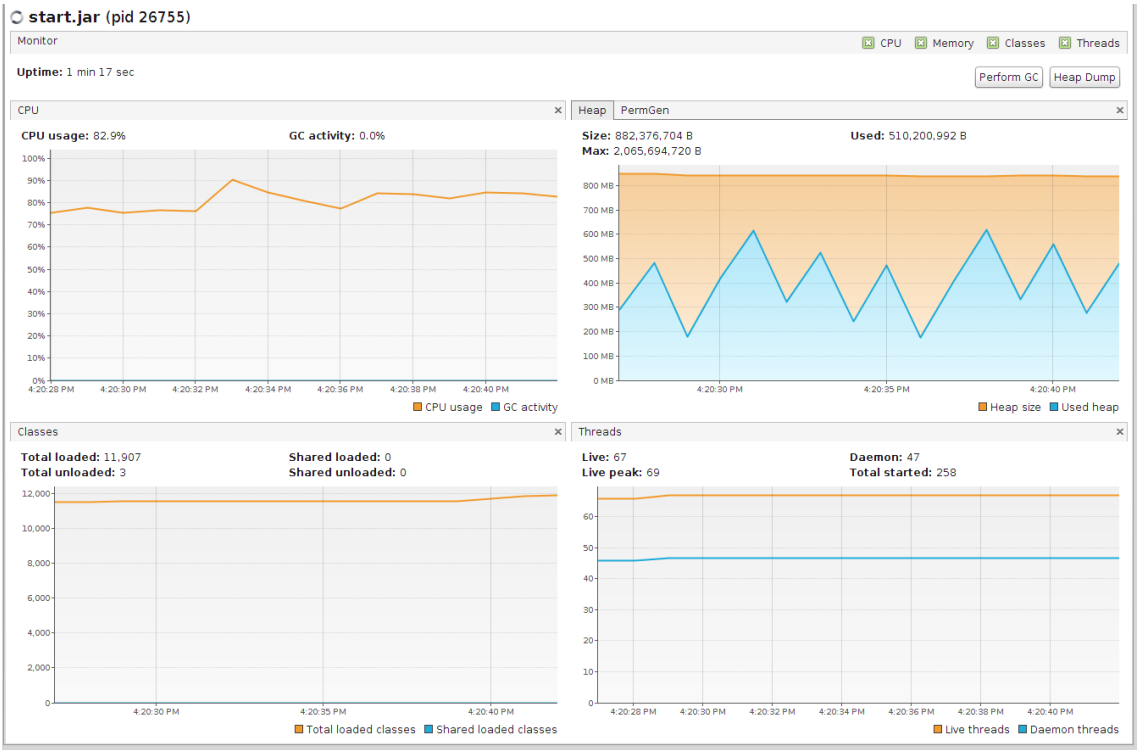

Figure 4.3: Resources used when running Exp6 with 4000 messages per minute

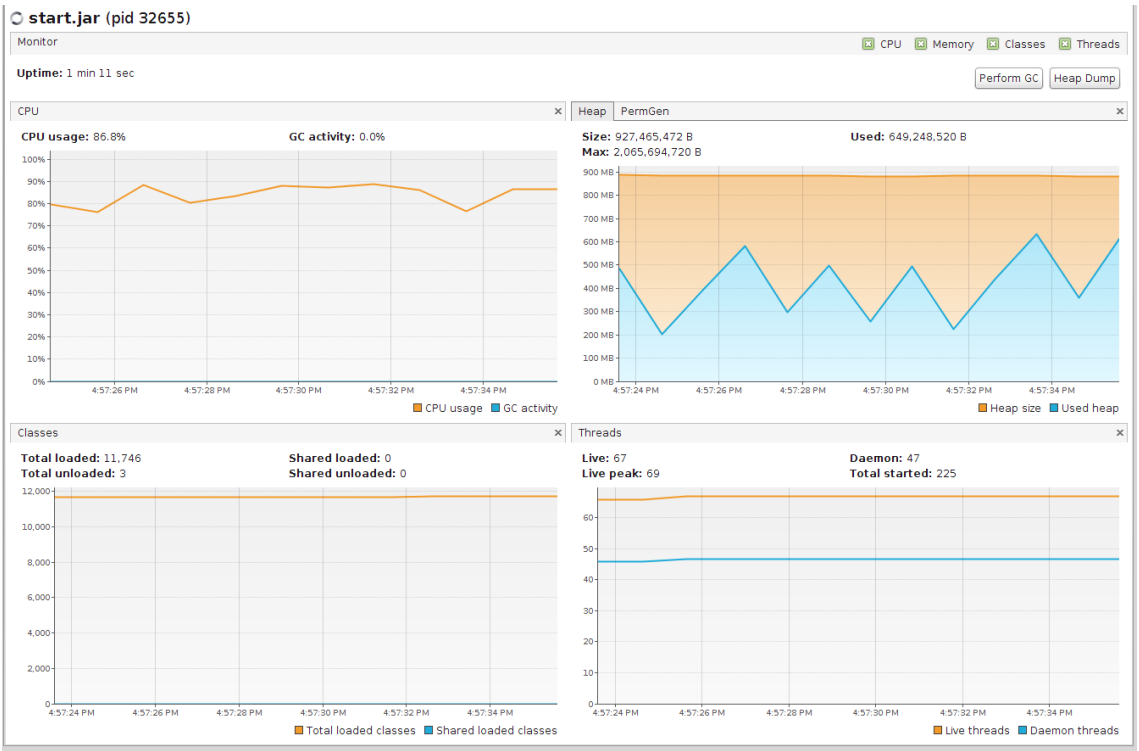

Figure 4.4: Resources used when running Exp7 with 4000 messages per minute

1 Source, 1 Session, 1 Client, Exp0			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	60	0	0.00
	60	0	
	60	0	
120	60	0	0.00
	60	0	
	60	0	
240	75	15	5.00
	60	0	
	60	0	
480	67	7	6.67
	66	6	
	67	7	
1000	114	54	55.00
	116	56	
	115	55	
2000	229	169	169.00
	229	169	
	229	169	
4000	459	399	397.67
	453	399	
	455	395	

1 Source, 1 Session, 1 Client, Exp6			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	87	27	27.00
	87	27	
	87	27	
120	89	29	29.00
	89	29	
	89	29	
240	89	29	29.00
	89	29	
	89	29	
480	89	29	29.00
	89	29	
	89	29	
1000	145	85	85.00
	145	85	
	145	85	
2000	257	197	199.67
	261	201	
	261	201	
4000	492	432	434.33
	496	436	
	495	435	

1 Source, 1 Session, 1 Client, Exp7			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	117	57	57.67
	118	58	
	118	58	
120	119	59	59.00
	119	59	
	119	59	
240	119	59	85.67
	119	59	
	199	139	
480	119	59	59.00
	119	59	
	119	59	
1000	174	114	113.00
	172	112	
	173	113	
2000	288	228	227.33
	288	228	
	286	226	
4000	516	456	458.00
	517	457	
	521	461	

Table 4.1: 1 source, 1 session, 1 client middleware execution times using SEDA

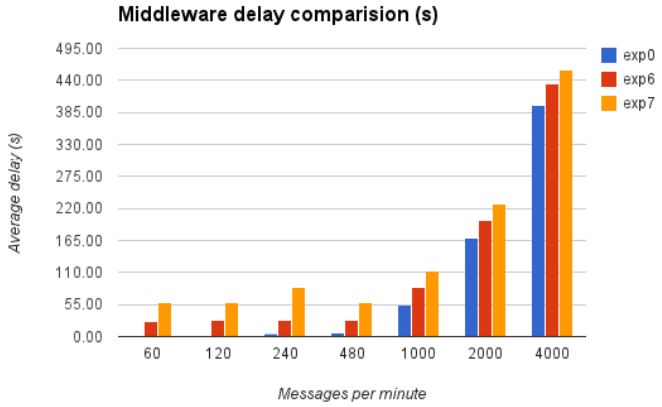

Figure 4.5: Middleware delay comparision

1 source, 2 sessions, 1 client, SEDA

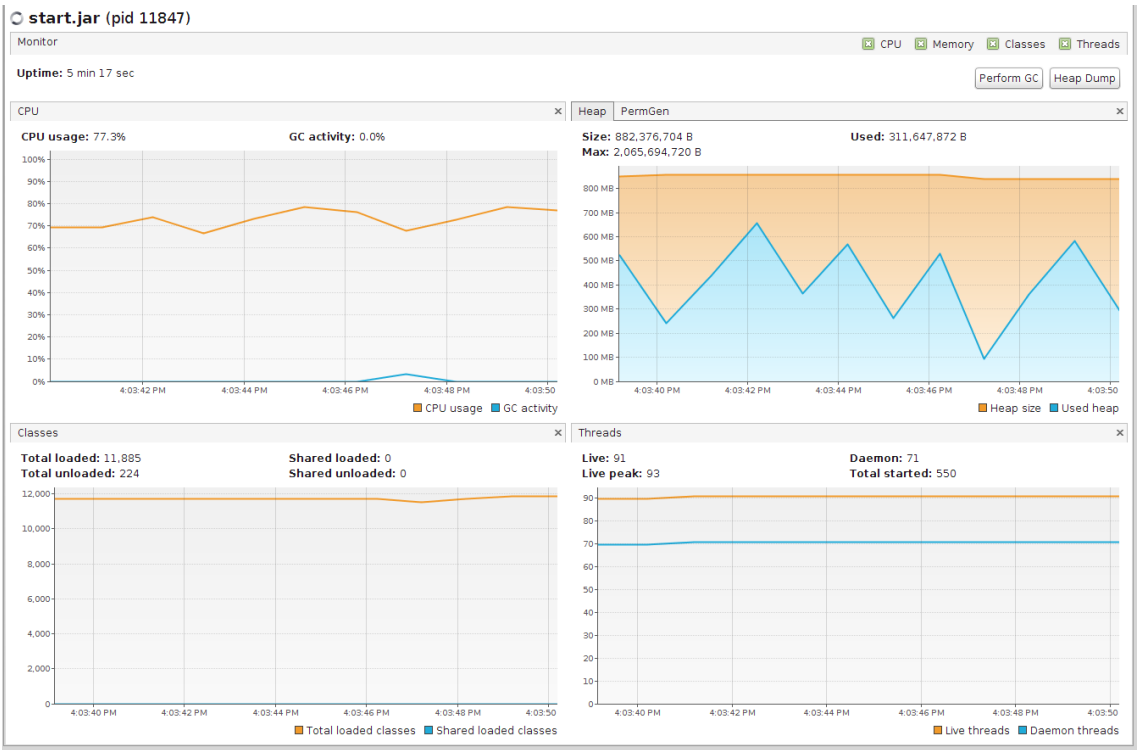

Figure 4.6: Resources used when running Exp0 with 4000 messages per minute

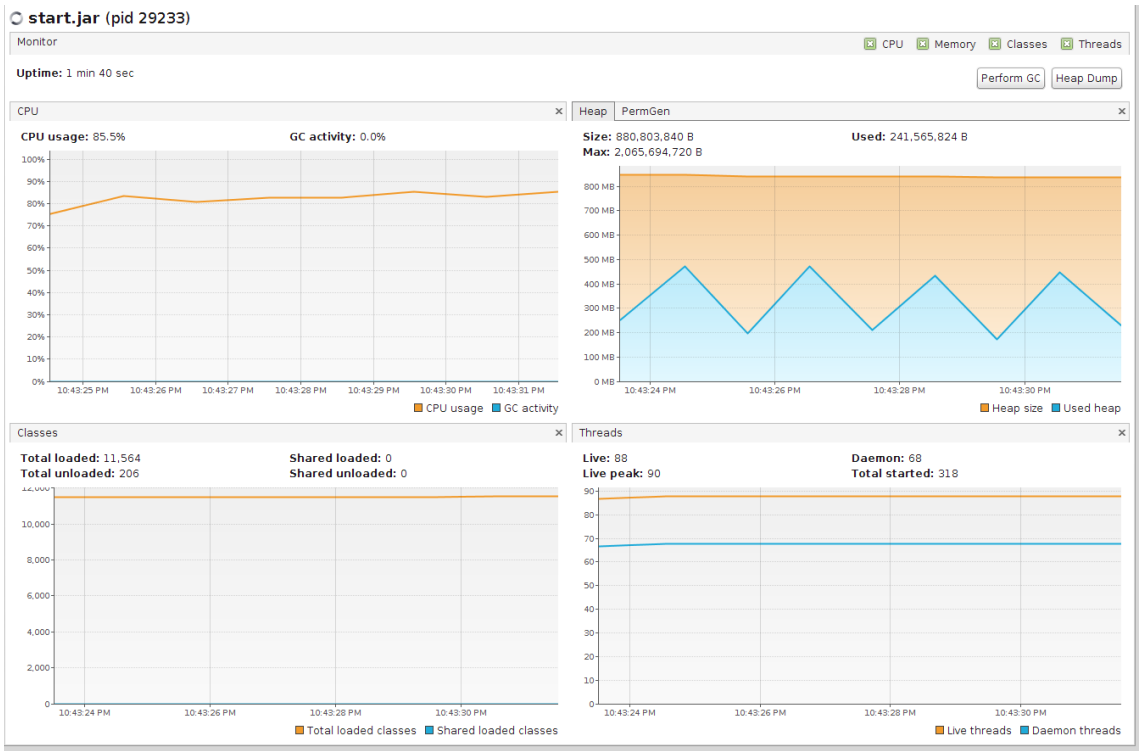

Figure 4.7: Resources used when running Exp6 with 4000 messages per minute

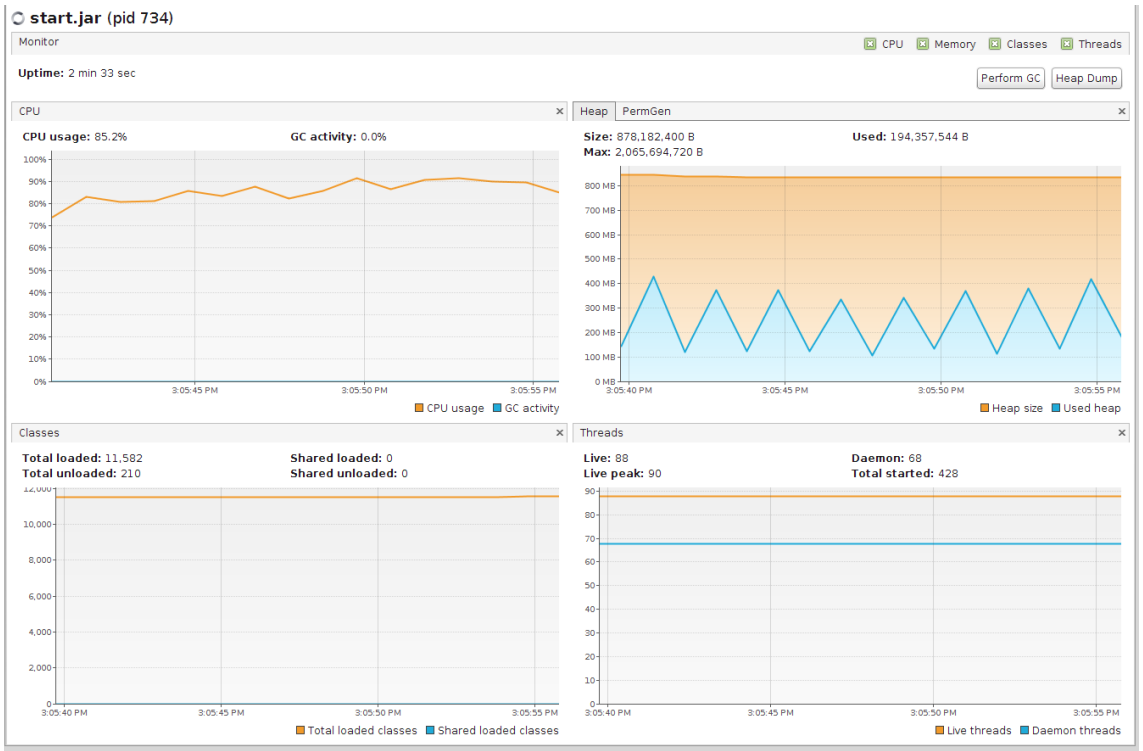

Figure 4.8: Resources used when running Exp7 with 4000 messages per minute

1 Source, 2 Sessions, 1 Client, Exp0			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	60	0	0.00
	60	0	
	60	0	
120	60	0	0.00
	60	0	
	60	0	
240	60	0	0.00
	60	0	
	60	0	
480	60	0	0.00
	60	0	
	60	0	
1000	151	91	87.33
	150	90	
	141	81	
2000	295	235	230.67
	299	239	
	278	218	
4000	551	491	518.33
	549	491	
	633	573	

1 Source, 2 Sessions, 1 Client, Exp6			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	87	27	27.33
	87	27	
	88	28	
120	89	29	29.00
	89	29	
	89	29	
240	89	29	29.00
	89	29	
	89	29	
480	101	41	39.67
	99	39	
	99	39	
1000	171	111	116.00
	181	121	
	176	116	
2000	318	258	257.33
	323	263	
	311	251	
4000	647	587	611.00
	686	587	
	719	659	

1 Source, 2 Sessions, 1 Client, Exp7			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	118	58	57.67
	117	57	
	118	58	
120	119	59	58.67
	118	58	
	119	59	
240	119	59	59.00
	119	59	
	119	59	
480	119	59	60.67
	124	64	
	119	59	
1000	199	139	142.00
	202	142	
	205	145	
2000	374	314	293.67
	350	290	
	337	277	
4000	624	564	586.00
	652	592	
	662	602	

Table 4.2: 1 source, 2 sessions, 1 client middleware execution times using SEDA

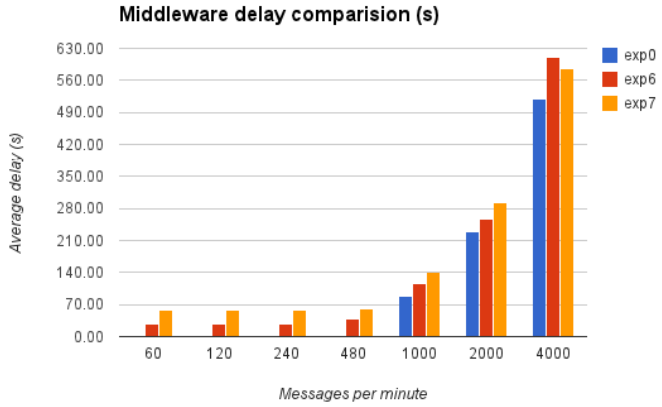

Figure 4.9: Middleware delay comparision

1 source, 4 sessions, 1 client, SEDA

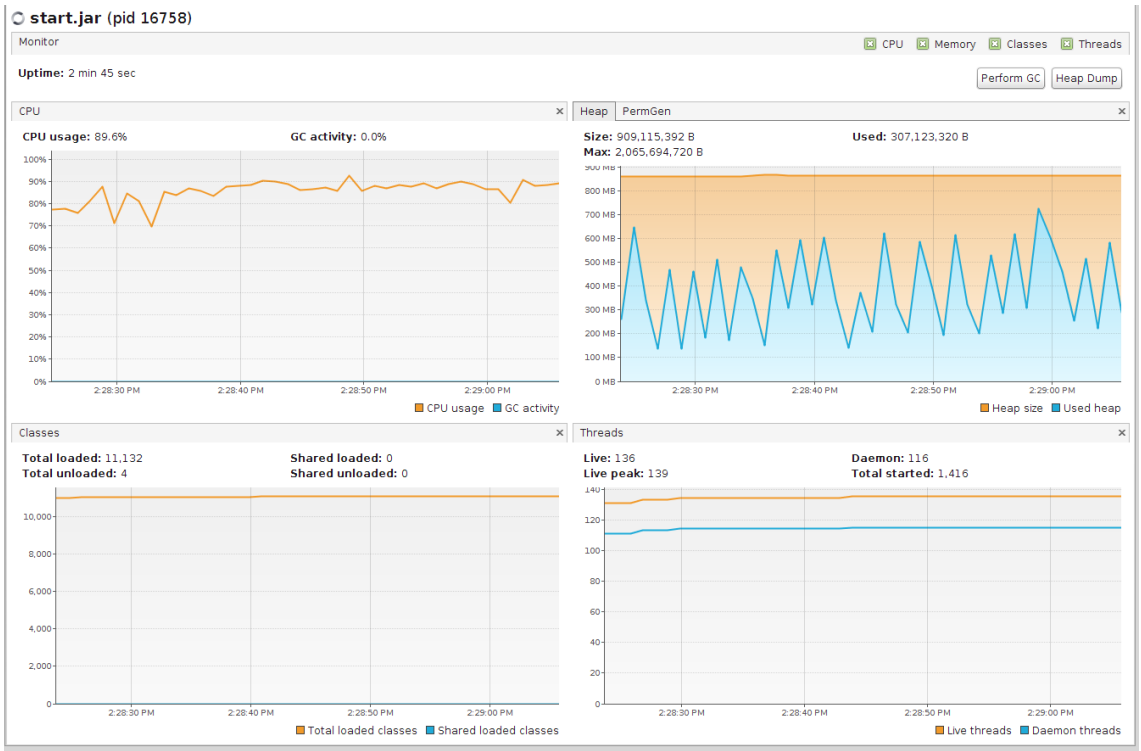

Figure 4.10: Resources used when running Exp0 with 4000 messages per minute

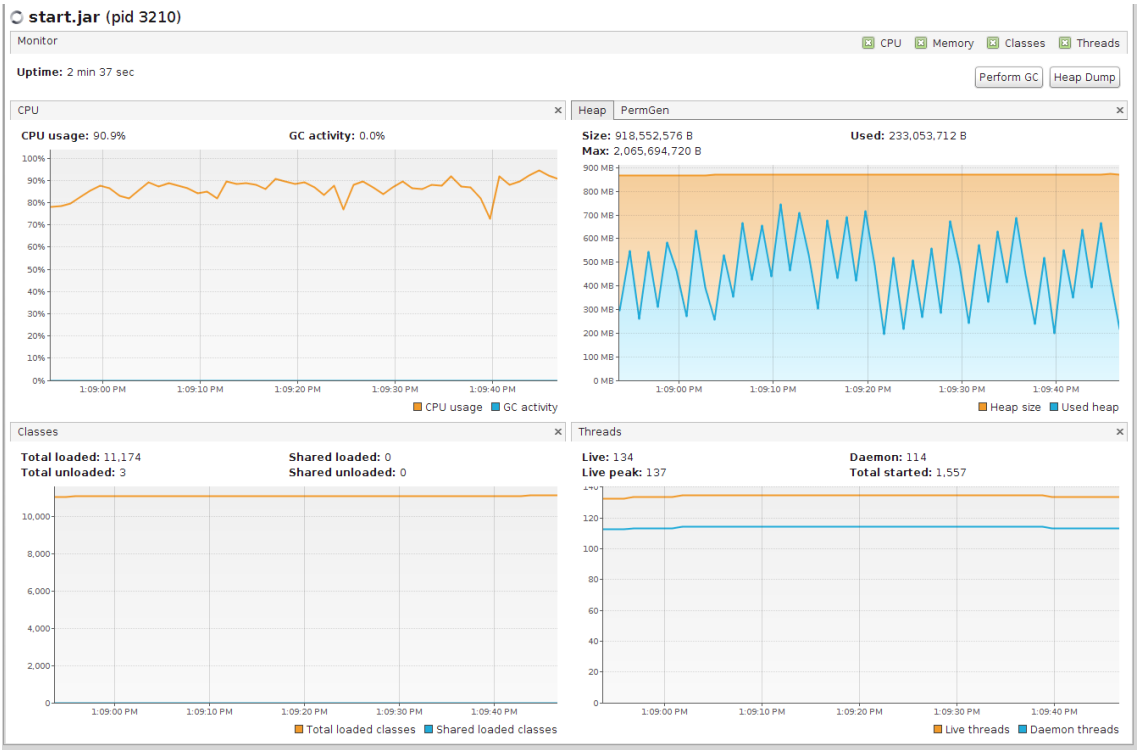

Figure 4.11: Resources used when running Exp6 with 4000 messages per minute

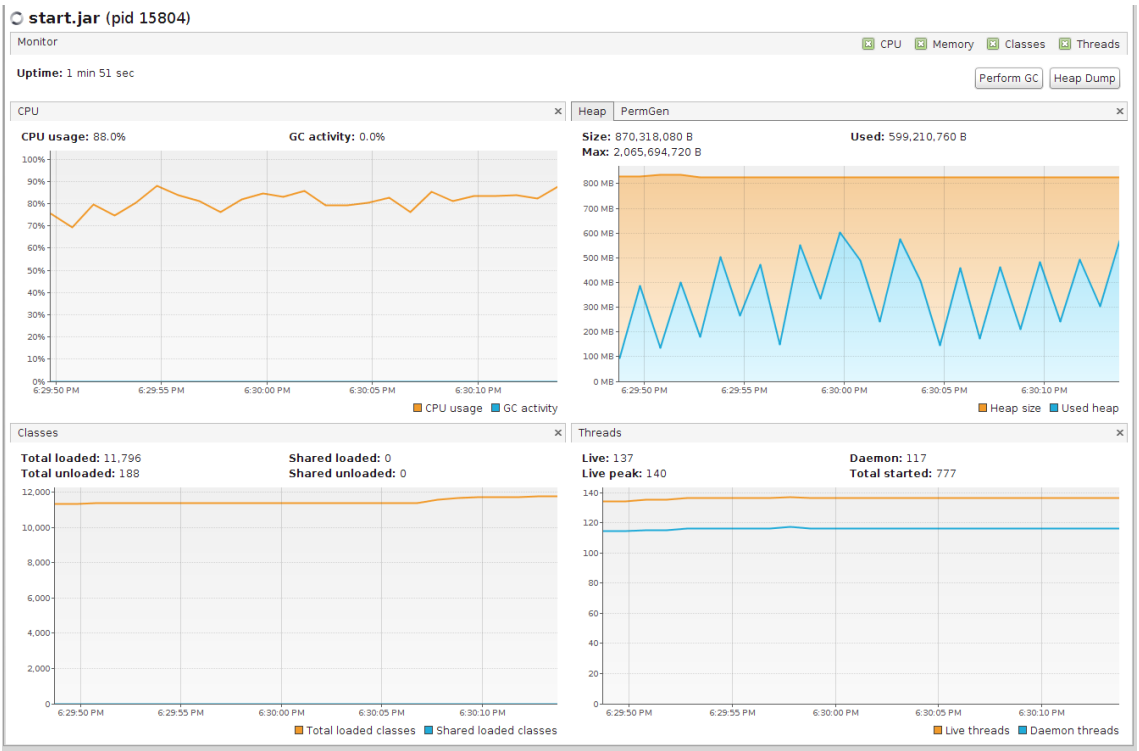

Figure 4.12: Resources used when running Exp7 with 4000 messages per minute

1 Source, 1 Session, 1 Client, Exp0			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	60	0	0.00
	60	0	
	60	0	
120	60	0	0.00
	60	0	
	60	0	
240	60	0	0.00
	60	0	
	60	0	
480	80	20	21.67
	84	24	
	81	21	
1000	172	112	107.00
	162	102	
	167	107	
2000	320	260	263.33
	332	272	
	318	258	
4000	629	569	563.33
	606	569	
	612	552	

1 Source, 1 Session, 1 Client, Exp6			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	88	28	28.00
	88	28	
	88	28	
120	89	29	29.00
	89	29	
	89	29	
240	89	29	28.33
	89	29	
	87	27	
480	120	60	58.00
	120	60	
	114	54	
1000	199	139	150.00
	225	165	
	206	146	
2000	421	361	342.33
	385	325	
	401	341	
4000	745	685	673.00
	726	685	
	709	649	

1 Source, 1 Session, 1 Client, Exp7			
Messages per minute per source	Middle-ware Exec. Time (s)	Delay (s)	Average Delay (s)
60	117	57	57.33
	118	58	
	117	57	
120	118	58	58.33
	118	58	
	119	59	
240	119	59	59.00
	119	59	
	119	59	
480	168	108	105.33
	162	102	
	166	106	
1000	260	200	200.00
	252	192	
	268	208	
2000	405	345	382.67
	450	390	
	473	413	
4000	870	810	821.00
	848	788	
	925	865	

Table 4.3: 1 source, 4 sessions, 1 client middleware execution times using SEDA

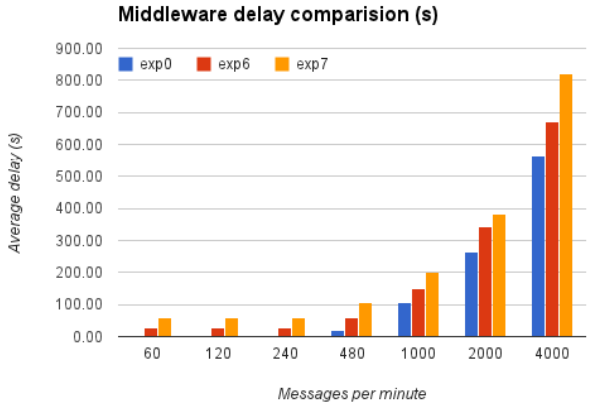

Figure 4.13: Middleware delay comparision

4.1.3.2 SEDA performance evaluation

As can be seen in Figures 4.2, 4.3 and 4.4 for the scenario with one session, Figures 4.6, 4.7 and 4.8 for the scenario with two sessions and Figures 4.10, 4.11 and 4.12 for the scenario with four sessions, the CPU usage ranges from 65% to 90%, which indicates that multiple cores are in fact being used in all the scenarios.

This is reflected in the middleware's execution times. For the scenario with one session running on 4000 messages per minute, seen in Tables 4.1 we won 114.66 seconds in Exp0, 108 seconds for Exp6 and 150.33 seconds for Exp7. For the scenario with two sessions, the gains were of 442.67, 117.33 and 122.66 seconds for Exp0, Exp6 and Exp7 respectively (see results in Tables 4.2) and for the scenario with four sessions the gains were of 917.34, 349 and 43.67 seconds for each expression respectively as well (see Tables 4.3).

In all cases the solution using DSL improved the middleware's execution time. This is better demonstrated by the graphs 4.14, 4.15 and 4.16, which directly compare the execution times of the middleware for the three expressions used, using DSL and not using DSL:

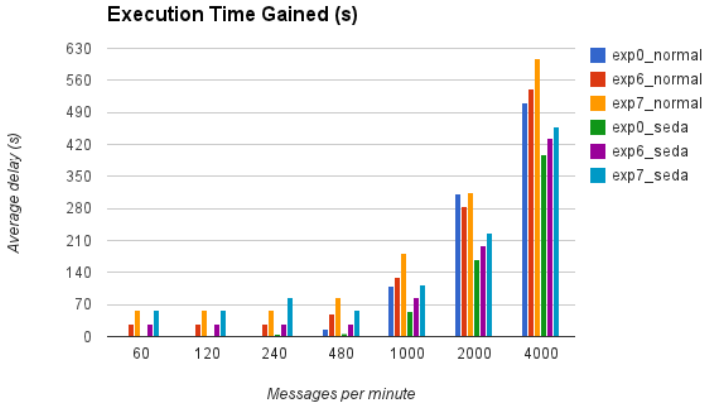

Figure 4.14: Timed gained by using the SEDA component in the scenario with 1 session

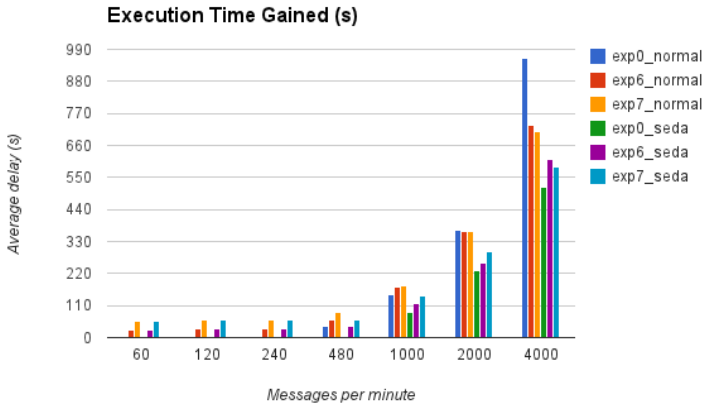

Figure 4.15: Timed gained by using the SEDA component in the scenario with 2 sessions

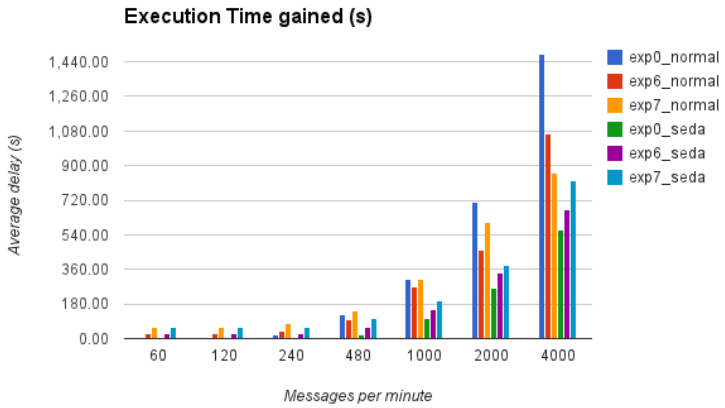

Figure 4.16: Timed gained by using the SEDA component in the scenario with 4 sessions

4.2 Data layer

In Section 3.2.2 we made a quick overview of the problems that using a RDBMS brings. In this section, we present NoSQL databases as a possible solution to fix those problems as well as the advantages they bring.

4.2.1 Why NoSQL?

As explained before, RDBMSs have limitations that make the scalability of the middleware harder. NoSQL addresses this problems by being:

- Easier to scale. NoSQL systems were made with scalability in mind and scale out instead of scaling up. In the cloud context, this means that we can have more power by using cheap machines instead of using a big one, which reduces the costs.

- NoSQL databases deal better with huge amounts of data. They are faster to retrieve results and can be more easily changed.

Furthermore, all of the previously studied CMPs support a NoSQL database in one way or another. This does not happen with the RDBMSs as we will see in Section 5.3, the level of support for key-value databases is simply greater. This means that deploying the middleware in a private cloud would be easier by using a NoSQL system instead of a common RDBMS.

Using a NoSQL database would also remove the Hibernate compatibility layer that maps RDBMS objects to Plain Old Java Object (POJO)'s, thus increasing the performance of the reads and writes done to the system because NoSQL system deal natively with such objects.

Unfortunately however, using a NoSQL system in the project would require massive changes to its structure, and a complete re-engineering of the core in order for it to work efficiently and effectively with the NoSQL system. Thus, in order to avoid remaking the structure of the middleware we propose a second alternative: the usage of a compatibility layer for NoSQL databases.

Such solution would give the middleware the main benefits of using a NoSQL system while still maintaining a decent compatibility level with the current state of the middleware. Therefore, the next sections will focus on finding and studying compatibility layers that can be used in order to add a NoSQL support to the middleware so it can use key-value stores such as the Amazon S3 widely adopted service.

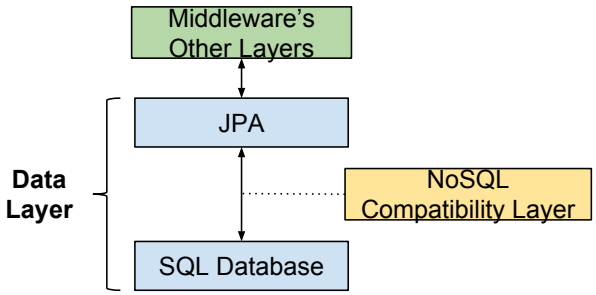

Figure 4.17: NoSQL compatibility layer

4.2.2 JPA compatibility for NoSQL

The main requirements for the JPA implementation that will be selected is the level of support for MySQL operations, namely the joins; if it supports Amazon's RDS service and if somehow it allows to convert data into a format that is compatible with Amazon's S3 service. Moreover, we will restrict the study to only the free and open-source implementations. As far as the NoSQL support goes, the chosen implementation must also provide a key-value store mapping, as that is going to be our main focus here.

4.2.2.1 Hibernate OGM

Hibernate OGM[10] stands for Hibernate Object/Grid Mapper. Announced in 2011, [Hig11], Hibernate Object/Grid Mapper (H.OGM)'s main goal is to provide JPA support for NoSQL solutions. Being a sub-project of Hibernate[11], H.OGM uses much of its father's building blocks, such as the Hibernate Core [Hibc], Hibernate Search [Gri12] and the JP-QL as a main querying language.

In order to support the key-value store front end, the project primarily used Infinispan[12] together with EHCache [13], however now it also supports MongoDB[14] and support for Voldemort[15] is also under way, while contributors from CouchDB[16] and Redis[17] are

[10] http://docs.jboss.org/hibernate/ogm/3.0/reference/en-US/html_single/#ogm-howtocontribute-contribute
[11] http://www.hibernate.org/
[12] http://www.jboss.org/infinispan/
[13] http://ehcache.org/
[14] http://www.mongodb.org/
[15] http://www.project-voldemort.com/voldemort/
[16] http://couchdb.apache.org/
[17] http://redis.io/

also considering investing [Hig11; Gri12].

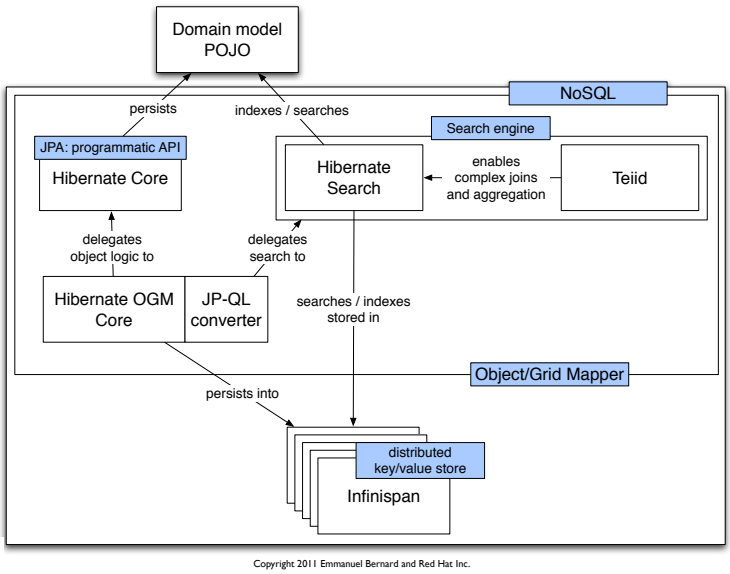

Figure 4.18: Hibernate OGM architecture

As far as H.OGM support goes, according to [Gri12; Ber11; BG11; Hibb] the following functionalities, limitations and future sights are provided:

Supports

- Object Oriented queries (JP-QL);
- CRUD of entities;
- Polymorphic entities;
- Embeddable objects (components);
- Basic types (some are not yet supported but can be trivially added);
- Unidirectional @ManyToOne, @OneToOne, @OneToMany @JoinTable, @ManyToMany;
- Bidirectional @OneToOne, @ManyToOne / @oneToMany;
- Collections (Set, List, Map, etc);
- Hibernate Search's full-text queries;
- JPA and native Hibernate ORM API support.

Limitations

- Does not support denormalization;
- Does not yet support complex joins and aggregation.

Future Objectives

- Develop high performance sequence generator;
- Apply parallel key fetching when possible;

- Add support for Map/Reduce;
- Support other NoSQL classes;
- Further mixing NoSQL with RDBMS.

Even though the project looks alive, the project's forum is quite dead, having most posts date back to the alpha stage of the project (announced in 2011) and most of the available information is also relative to that period. Furthermore, the documentation is scarce and greatly scattered throughout the Internet. While the official documentation is indeed updated to comply to the beta stage of the project, it is very small, and does not cover all the project's limitations and strengths, limiting itself as a guide on how people can contribute and on the most common settings of the project. As far as support goes, there is commercial support provided by Red Hat to the project [Hiba], an in fact most of the solutions found using H.OGM, were based on Red Hat products, liable to Red Hat commercial support.

4.2.2.2 EclipseLink NoSQL

EclipseLink NoSQL[18] is a sub-project of EclipseLink (here known as the father project) which was started based on TopLink, an Oracle's product [Ora07]. Because EclipseLink NoSQL is strongly connected to the father project, its almost impossible to talk about one without mentioning the other. The father project's goals, are to provide a complete persistence solution, capable of running in any Java environment and with ability to write and read objects to and from any kinda of source, be it relational databases, XML or EIS. The father project implements support for JPA, to deal with relational databases; JAXB, to bind Java and XML; JCA, for EIS and other types of legacy systems and SDO [Ecla].

It was this main desire of providing a complete persistence solution that led to the creation of EclipseLink NoSQL. The son project adds NoSQL support to the father, by supporting NoSQL databases such as MongoDB, Oracle NoSQL as well as other services like Oracle AQ, JMS and XML files [Ecld; Eclb].

As far as the architecture goes, like with H.OGM, EclipseLink NoSQL uses much of what the father has to offer. Although it requires an EISPlatform in order to access NoSQL sources, it provides NoSQL platforms for MongoDB, Oracle NoSQL, XML file, JMS and Oracle AQ. NoSQL support is also flexible, it is built on the top of the JCA father's component, but it can also support third party JCAs provided that they comply with the JCA CCI API [Ecld; Eclc; Eclf].

[18] http://www.eclipse.org/eclipselink/downloads/

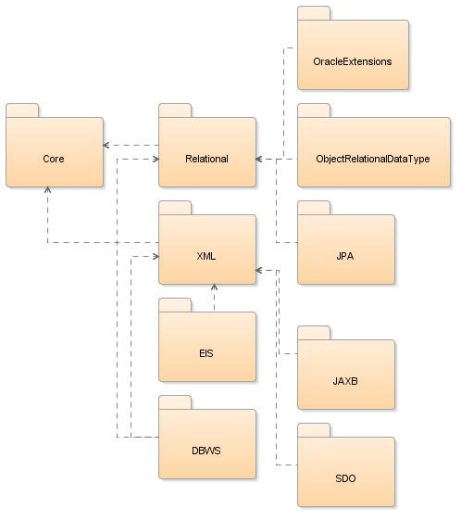

Figure 4.19: EclipseLink architecture and how EclipseLink NoSQL connects

EclipseLink NoSQL supports also a variety of functionalities as described bellow [Ecld; Eclc; Eclf]. However, the team behind EclipseLink NoSQL did not make public any future plans and there is little to no documentation about the limitations of the project. This is in part due to the fact that great part of the documentation is made by the community, and so it is difficult to guess what future developments each member has in mind, and which limitations each one is trying to fix. Still, EclipseLink NoSQL benefits from a better documentation than H.OGM, more organized, with more examples [Ecle], which greatly comes from the father project.

Supports

- Object Oriented Queries (dependent on the NoSQL platform);
- Polymorphic entities;
- Basic types;
- Unidirectional relationships;
- Collections;
- Most of JPA - some features such as joins, atomic transactions are not supported if the NoSQL platform does not support them;
- Complex hierarchical (including XML);
- Indexed hierarchical data;
- Mapped hierarchical data (such as JSON);
- CRUD operations;
- Embedded objects and collections;
- Inheritance;
- Subset of JP-QL and Criteria API, dependent on NoSQL database's query support;
- Denormalization.

Limitations

- Joins are not supported. Queries to embedded relationships is.

4.2.2.3 DataNucleus

Unlike the previous two sub-projects studied, DataNucleus[19], formally known as JPOX (Java Persistent Objects), is a project on its own. It advertises itself "(...) as the most standards-compliant open-source Java persistence product in existence (...)" [Data].

This statement is likely to be true: DataNucleus is fully compliant with the latest versions of the JPA and Java Data Objects (JDO) APIs and it also complies with the Open Geospatial Consortium (OGC) Simple Feature Specification for persistence of geospatial Java types to RDBMS as well as REpresentational State Transfer (REST) [Data; Bas11].

As far as data-store formats go, it supports Google Big Table, MongoDB, Cassandra[20], a wide range of RDBMS databases [Datf], Excel, OOXML, ODF, XML, HBase[21], AppEngine/DataStore, Neo4j[22], JSON, Amazon S3, GoogleStorage, LDAP, NeoDatis[23] and db4o[24] [Bas11; Datd; Datb].

All this flexibility is only possible due to the Open Service Gateway initiative (OGSi)-based plugin mechanism DataNucleus implements, which allows for anyone to build compatibility plugins for the platform [Data; Datg]. This plugin system is deeply integrated in the very simple architectural model of the platform, and it's use by the community has been seen mainly in the store management component [Datc].

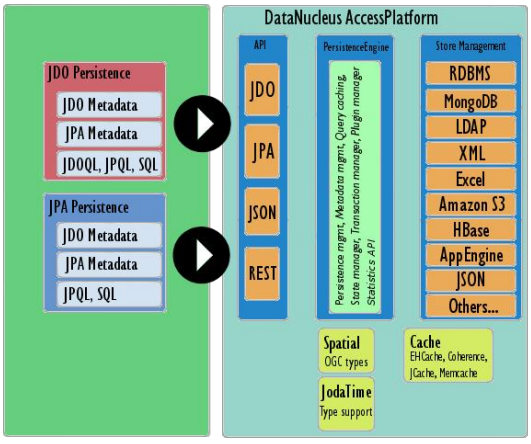

Figure 4.20: DataNucleus main architectural components

Thanks to all the previous supported datastores and APIs, DataNucleus has a long list

[19] http://www.datanucleus.org/development/scr.html
[20] http://cassandra.apache.org/
[21] http://hbase.apache.org/
[22] http://www.neo4j.org/
[23] http://neodatis.wikidot.com/
[24] http://www.db4o.com/

of supported features, to big to fit here. Still, for the sake of coherence, here is provided a small list of supported features that conforms with the previously seen lists. The full list of features can be seen in DataNucleus's datastore features web page[25]. Furthermore, unlike EclipseLink NoSQL, DataNucleus is more open to the community, making its future plans known [Date] and actively engaging in a forum that is more responsive than EclipseLink NoSQL and H.OGM forums. This is probably derived from the big community, actively engaging in the creation of additional plugins and resources for the project and also due to the fact that DataNucleus also offers commercial support, given through the forums and other means of communication [Datg]. This is therefore reflected in the documentation, that is well organized, small and direct to the point. However, most of the important documentation found is through personal blogs, websites and tutorials, that although scattered throughout the Internet, are still very easy to find and overall informative.

Supports

- CRUD operations;
- Embedded objects and collections;
- Inheritance;
- Relationships (Unidirectional and Bidirectional);
- Queries for JP-QL, JDOQL and SQL (partial);
- Basic types;
- Joins.

Limitations

- Aggregations? (not specified in documentation).

Future Objectives

- JPA2.1 full feature list
- Official support for Cassandra
- Consider a plugin for REDIS

4.2.2.4 JPA summary

In this section we tried to compare the most common JPA implementations with NoSQL support. This is a complex topic because most of the existing information is focused on their JPA implementations and not in the NoSQL support. These chapters therefore focused on comparing functionality and support, rather than comparing benchmarks - a topic that needs to be studied by community with more detail.

H.OGM is supported by Red Hat and uses much of the father's components. It also uses other Red Hat free components, such as Infinispan, as a main key-value store, but it also supports MongoDB and work to support more functionality is on the way. Of all the

[25] http://www.datanucleus.org/products/accessplatform_3_2/datastores/datastore_features.html

three platforms studied here, this is the one whose objectives, limitations and goals are more detailed and explicit. This is due to the fact that its development is more centralized, and less dependent on a community.

EclipseLink NoSQL supports more features than H.OGM and more datastores. While H.OGM and the overall Hibernate project are highly connected to Red Hat, EclipseLink NoSQL and EclipseLink are tied to Oracle and its contributions. In this case the documentation is a lot better than the Hibernate's documentation and it also identifies itself as being faster than it. However, EclipseLink's support system is not very good. Every post in the forums requires an approval period that can last for several days (thus delaying the answers to the problem). Furthermore, unlike Hibernate that provides commercial support through Red Hat, or DataNucleus that has its own commercial support team, EclipseLink has no commercial support feature. In fact, the nearest thing to commercial support that EclipseLink has is using Oracle's TopLink[26] service, which encapsulates EclipseLink with many other proprietary Oracle products [Kar10].

Last but not least, is DataNucleus. Thanks to its plugin system, this platform has a massive support to nearly all kinds of datastores, winning against H.OGM and EclipseLink NoSQL by a large amount. As far as documentation goes, the project has rich documentation, although not as good as EclipseLink's documentation, it still beats Hibernate's by a long shot. Furthermore while Hibernate and EclipseLink are tied to Red Hat and Oracle respectively, DataNucleus is independent - and it still offers commercial support, which can be acquired in the forums that are more active than the forums of the previous projects.

[26]Oracle TopLink project - http://www.oracle.com/technetwork/middleware/toplink/overview/index.html

	Hibernate OGM	EclipseLink NoSQL	DataNucleus
Goal	Complement JPA with NoSQL, key-value stores	Integrates in the father project main goal of providing a complete persistence solution	Being a standards compliant and efficient JPA and JDO platform
NoSQL and Datastores supported	Infinispan, EHCache, MongoDB	MongoDB, Oracle NoSQL, Oracle AQ, JMS, XML files	Google Big Table, MongoDB, Cassandra, Excel, OOXML, ODF, XML, HBase, AppEngine/DataStore, Neo4j, JSON, Amazon S3, GoogleStorage, LDAP, NeoDatis, db4o
Operations supported	Object Oriented queries (JP-QL), CRUD of entities, Polymorphic entities, Embeddable objects, Basic types (partial), Unidirectional and Bidirectional relationships (partial), Collections, Hibernate Search queries, JPA and Hibernate ORM API	Object Oriented Queries, Polymorphic entities, Basic types, Unidirectional relationships, Collections, JPA (partial), Complex hierarchical, Indexed hierarchical data, Mapped hierarchical data, CRUD operations, Embedded objects and collections, Inheritance, Subset of JP-QL and Criteria API, Denormalization	CRUD operations, Embedded objects and collections, Inheritance, Relationships (Unidirectional and Bidirectional), Queries for JP-QL, JDOQL and SQL (partial), Basic types, Joins.
No support for	Denormalization, Complex joins and aggregations	Joins	Aggregations? (not specified in documentation)
Future	High performance sequence generator, parallel key fetching, support for Map/Reduce, more NoSQL classes, better mixing of NoSQL and RDBMS	?	JPA2.1 full feature list, Official support for Cassandra, Considering a plugin for REDIS
Commercial support	Red Hat	Oracle (via TopLink)	Supported by DataNucleus team
Documentation	Scattered, inactive forums, official documentation lacking	Bureaucratic forums, information is complete and gathered mainly in the official website	Active forums, acceptable official documentation, but the big advantage comes from user support in form of blogs and posts scattered around the Internet

Table 4.4: JPA compatibility features comparision

5

Elastic Cloud Deployment

Chapter 4 addressed the optimization of the middleware within a single computing node, by exploring how the usage of additional threads and Apache Camel components could affect it. In this chapter we will tackle scalability issues, presenting solutions for how to make the middleware scale out int the cloud.

5.1 Scalability in the Cloud

The solutions proposed in Chapter 4 leverage the parallelism available in current multi-core CPUs to explore all the capabilities of a single machine. However, if there are enough data-sources, sessions or clients, the machine will eventually reach its maximum usage, and thus it will become a bottleneck due to the shortage of resources.

To tackle this problem the following two solutions were considered:

1. Having the middleware scale up
2. Having the middleware scale out

The first option simply does not scale well enough because it gets very expensive very fast, even when the VM in cause is being shared. The second option however, seems to be more feasible. To solve the problem of scalability, one could create additional VMs when needed, and then kill them when no longer needed. This alternative is not only cheaper than the previous one, it also allows the deployment of the middleware in weaker machines.

For the second solution to be feasible however, one would need to have a monitoring system in the current machine, and a way to create additional machines. The monitoring

system could take into account multiple factors, such as the CPU load and memory begin used, and then use an API to ask for the creation or destruction of new machines. Furthermore, the management of such machines could follow well known strategies, such as the master and slave strategy or the less centralized peer to peer strategy.

5.1.1 Master and Slave

In this scenario, a master is a regular VM running the middleware, but it receives all the requests from the clients, redirecting them to the proper slaves when needed. The master does not have the capability of creating or destroying other VMs, instead it delegates that capability to a load balancer. The load balancer is another VM, solely devoted to the purpose of collecting status information from all the VMs currently active and the only one capable of creating and killing them. Because the load balancer has information about all the instances currently active, it is capable of deciding when to create or kill instances. Furthermore, it also knows which slaves are free to take requests from clients, and so it aids the master when selecting a slave when a request comes.

Decoupling the master from the load balancer frees the master from the problem of becoming overwhelmed by slave requests, because all the slaves communicate with the load balancer instead. Furthermore, because the responsibilities are divided, the architecture is also more modular.

This solution is, however, not without drawbacks. Its centrality raises reliability issues. The master is a single point of failure because if it dies no one is left to coordinate the slaves nor manage their work load by the creation or deletion of additional slaves. The problem may be overcome with a fail-over mechanism, which monitors the execution of the master via a beat-heart protocol, and an election algorithm, should there be a need to elect a new master from the slaves, or to create a new one from scratch.

Another setback for this solution is the load the clients may impose upon the master. As the number of client requests grows, the master eventually becomes too occupied redirecting them, thus compromising the execution of the sessions assigned to him. The reverse is also possible - if the sessions assigned to the master demand to much, new client requests may be put on hold. A solution to this problem is to turn the master into a dedicated server, with the sole purpose of only redirecting clients once a certain threshold is hit, and then go back to becoming a regular server once the work load goes back to normal levels. To work perfectly, this solution will require session migration as well as client redirection and data-source redirection, and none of these is currently implemented nor supported by the middleware. Furthermore, the load balancer would also have to be prepared to treat the master as a special case, allowing it to become dedicated.

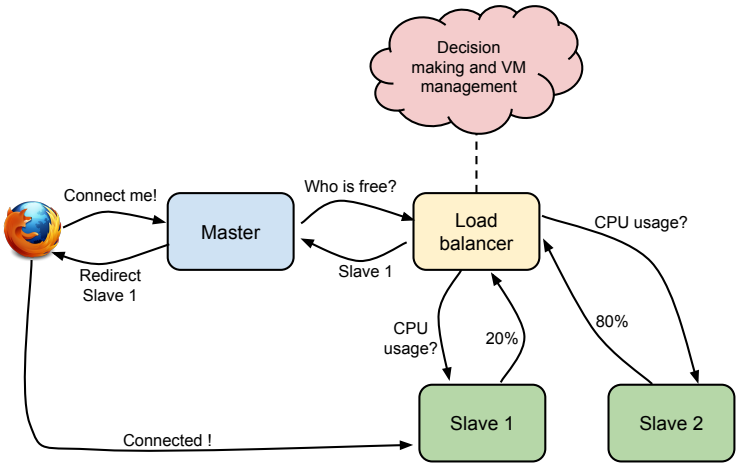

Figure 5.1: Master and Slave design

5.1.2 Peer to Peer

The alternative peer-to-peer approach is a decentralized solution. In this scenario, a peer is a VM running the middleware with the extra layers needed for peer coordination.

In this solution, each peer can check the current work load either when receiving a request or periodically, and then with this information decide if it should take the new request or not. If not, then the peer forwards the request to its neighbors. If no neighbors exist in its vicinity, then the peer creates them. In this solution, each peer is a mini master, with the power to create additional instances, but not to destroy them. The removal of a machine will have to be a suicide call, after warning all the neighbors and migrating any information needed. To achieve this, a modification of the Chord Algorithm [SMKKB01] can be used, but there are many other peer to peer strategies that can be interesting as well, such as the ones used in Pastry and Tapestry [Sem], or even the KaZaa [GK03] and Skype [MR06], [GDJ06] algorithms which use super-nodes.

When compared to the master and slave approach, a distributed solution will alleviate the work load on the VMs and will make VM failures less significant because this way only part of the sessions will be lost, instead of losing all the sessions in the master plus all coordination. In fact, since each peer will theoretically be independent from the others, one will not even need a beat-heart algorithm to check for signs of life because there are no masters that hold the management of creation and deletion of VMs. The only drawback to this solution, besides the migration support that the middleware does not yet have, will eventually be the complexity inherent the peer network ring and the coordination it requires between the peers.

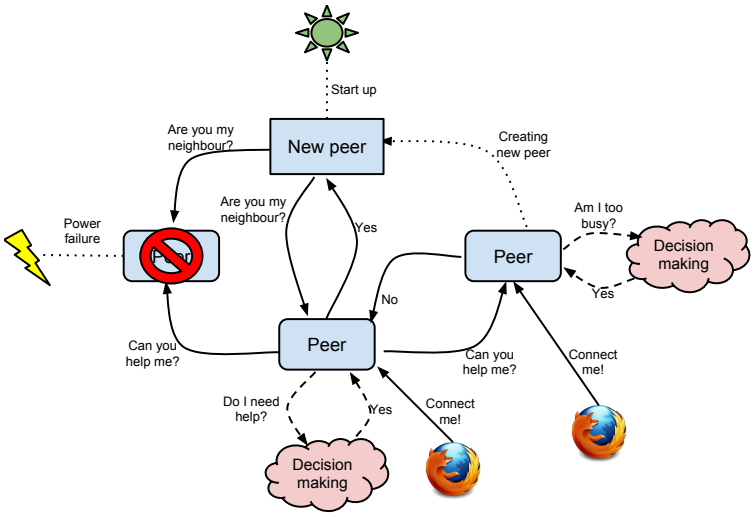

Figure 5.2: Peer to peer design

5.2 Implementation details

In Section 5.1 we addressed the scalability problem of the middleware and proposed two solutions for it. In this Section we explain how the Master and Slave proposal from Section 5.1.1 was partially implemented.

Hence, we start this section by giving a brief introduction of the differences between the proposed architecture and the implemented one. Then we continue by explaining how we monitor the each instance's resources, which tools we used and how we used them. We move to VM management and communication where we introduce the reader to he communication protocols used as well as to the algorithm that decides how and when new instances are created and terminated. We end this section with an overview of the current limitations of the implementation and how some of them can be addressed or minimized.

5.2.1 Architecture

The implementation of the master and slave strategy is only a proof of concept, mainly focused in the VM management. Here the master and the load balancer are not decoupled, they are embedded into the master, which takes all the decisions regarding VM creation and deletion while still taking client requests.

In this version, each slave also has the capability of monitoring its own state, and it decides to ask the master to create a new VM or to commit suicide depending if it is being stressed or if it is idle. Even though the slaves have more autonomy, it is the master who ultimately decides to accept the request or not. This is a important feature that does not fully compromise this proof of concept into the master and slave strategy only. With a few modifications, namely the introduction of the ring algorithm used in Chord, the slaves

would be practically independent from the master and thus would work as autonomous peers.

There is however, an important detail that is lacking - currently the master does not yet redirect the clients to the idle slaves. As a consequence, the clients have to connect themselves directly to the instances that they think may be idle. This will work fine if this implementation is to be turned into a peer to peer system, however, if in the future the master and slave strategy is preferred, redirection must be implemented so the master can redirect clients to other slaves.

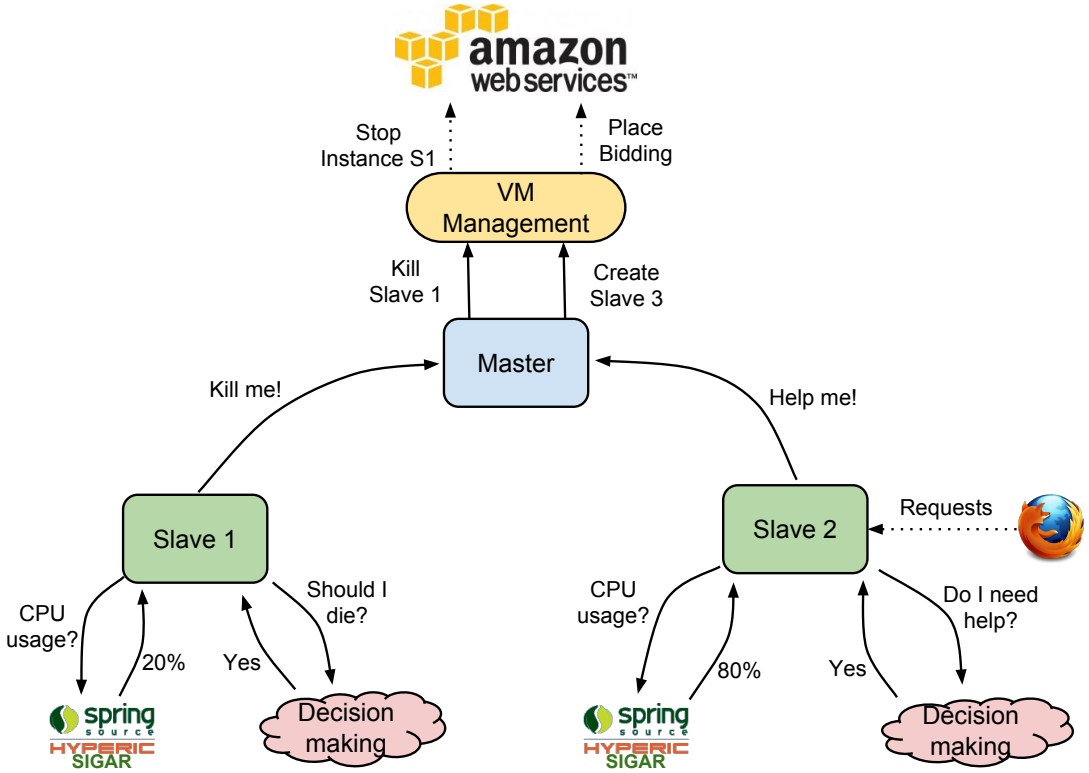

Figure 5.3: Architecture of the current implementation

5.2.2 Monitoring instance state

The first step in the implementation of the Master and Slave solution was to find an API or library capable of monitoring the current state of a regular VM. For this purpose, we have chosen SIGAR[1] because it allows us to access system information regardless of the operative system being used. SIGAR is versatile in that it also allows us to retrieve the CPU load, memory being used, network traffic and many more things, so it fits our needs perfectly.

[1] http://www.hyperic.com/products/sigar

Thus, when the middleware is launched, it starts a service that runs periodically. This service, checks for the CPU levels every thirty seconds, and when it does so it adds the result of the current check to an history list containing the last ten results.

When a user needs to create a session, the history list is checked, and an average of all its values is calculated. Everything explained so far works in all the VMs equally, independent of their role. However, the decisions taken based on the value of the average are different, depending if the VM running the middleware is a master or a slave.

5.2.3 VM management and communication

Before anything else, a newly created VM must know if it is a master or a slave. This is currently done by comparing the public IP of the newly created VM to the public and elastic IP of the master. The master has one IP that never changes, even if the instance is restarted or shut down. If the IP is different, the VM is a slave, if its not, then its the master.

Back to managing the amount of VMs when creating a session, if the VM is the master and if the value of the calculated average is big enough (meaning the master is overwhelmed with work), then the master directly uses the AWS Java SDK and communicates with the EC2 service in order to place a bid for a m1.small instance type. If the VM is the master and the value of the calculated average is very low (meaning that the instance is currently without work), then nothing happens. If no VMs are running the middleware then the service is dead, and that makes no sense, so there must always be one instance running no matter what, and that instance must be the master.

If the VM is a slave and the value of the calculated average is too high (meaning the slave is overwhelmed with work), then the slave asks the master to create a new instance. If the value is instead too low, then the slave asks the master permission to die, and if the master decides that the slave has outlived its usefulness, the master terminates it.

All the communication between master and slaves is done via Java Remote Method Invocation (RMI). The master VM provides the following interface that allows the slaves to make requests:

Listing 5.1: Thread B carrying information from the second Esper endpoint server websockets

```
1  package net.jnd.thesis.service.loadBalancer;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  /**
7   *
8   * Interface that exposes the methods offered by the RMI server deployed in the
        Master.
9   */
10 public interface ICommunicator extends Remote{
```

```
11
12    /**
13     * Ask for a new VM.
14     *
15     * @return          <code>true</code> if the request was successful,
16     *              or <code>false</code> if it failed or was denied
17     * @throws  RemoteException if this EC2 instance is a son and there was a
18     *              problem communicating with Dad
19     */
20    public boolean askNewVM() throws RemoteException;
21
22    /**
23     * Kill target VM.
24     *
25     * @param anInstanceId  the id of the EC2 instance we wish to terminate
26     * @return          <code>true</code> if the request was successful,
27     *               or <code>false</code> if it failed or was
28     *              denied
29     * @throws  RemoteException if this EC2 instance is a son and there was a
30     *              problem communicating with Dad
31     */
32    public boolean killVM(String anInstanceId) throws RemoteException;
33  }
```

As can be seen, a slave can either ask for a new slave, or ask to kill a VM. As of this point, each slave can only ask the master to kill himself, so this method is always used with the id of the VM making the call. However, since in the future this may change, by adding extra functionality to the slaves in order to make a more peer to peer solution for example, the decision of making the interface ready for that was taken.

5.2.4 Limitations

First of all, placing a bid to request an AWS instance is a dual edged sword. In one hand we can get an instance running for a very cheap price, in the other hand, if we bid incorrectly we may have to wait a very large amount of time until we can actually get a instance with the bidding price set. A solution to avoid this would be to just reserve a machine and pay the recommended price for a m1.small instance, however, given the fluctuations in price, bidding for machines can become quite profitable if one is willing to take the risk and wait a little.

Furthermore, and as previously stated, all the communication between the master and its slaves is done via the Java RMI communication protocol inside the Amazon cloud. This works fine if the middleware setup is deployed inside the same network, however RMI is blocked by external routers, so it is not possible to have two VMs communicating if they are in different Amazon regions, such as Europe and America. And even when inside the same country, there are regions, for example, North America is divided into one US East center (N. Virginia) and two US West centers (Oregon and N. California), which

cannot communicate with each other using RMI. The only way to manage communication from region to region is to use AWS services and upload data from one region to another, paying the respective upload and download fees for moving that data. This limitation can be addressed by configuring RMI through firewalls via proxies[2], by converting the requests to HTTP. However this solution requires a proxy, the speed of transmission is at least an order of magnitude slower, and the client cannot export its own methods outside the firewall.

Then there is also the problem of VM identity via elastic IPs. Though admittedly this solution works perfectly for the current scenario, it does raise two problems:

1. When the instance is shut down, AWS forces the users to pay an extra fee for having the elastic IP reserved but not using it;

2. If the instance is terminated, a new elastic IP has to be addressed.

While the first problem can be minimized by having the least amount of down time possible, the second problem is more complex. Addressing a new elastic IP is currently a problem because the public elastic IP of the master is hard coded. Thus, if it changes, the code must be recompiled and new Amazon Machine Image (AMI)s have to be created and deployed in the Amazon so the slaves know which master to contact. This is a workaround to avoid the implementation of an election algorithm on start up, but it can be costly in terms of time.

Last but not least, although the newly created VMs can accept new requests, due to the fact of not having any mechanism of data and information migration, the requests from the old VMs are not yet redirected to them. This means that the old VM will still be running at full capacity, even though a new slave has been created.

5.3 Deploying on Private/Hybrid Cloud Management Providers

The current version of the middleware is deployed in the Amazon cloud, using the EC2 and Elastic Block Store (EBS) services, and it was already deployed using EC2, EBS and RDS. This incurs into a monthly cost that depending on the scale of usage, may not be supported by the administrator of the system.

To solve this problem or at least help attenuate some of the costs, some administrators may choose to deploy the middleware in a private cloud, or in a hybrid could. This could allow for lower costs, and it would grant more autonomy over the data being stored and transfered. This would be extremely beneficial for those companies that provide services containing sensitive information, like hospitals and banks.

Although using encryption on the cloud can be a solution, such is not practical at all because the encryption algorithms and strategies are heavy and impractical. Furthermore the few solutions of cloud encryption that exist are not complete and bullet proof.

[2] http://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-arch6.html

Therefore, these reasons make viable to conceive a scenario where a part of the processing or storage has to be done outside the cloud. But what to choose then? Wat CMPs could the reader use to make a private or hybrid cloud? Currently the market has a considerable number of open-source CMPs. Having this in mind we considered and analyzed the following:

- Eucalyptus;
- OpenStack;
- OpenNebula;
- Nimbus;
- CloudStack.

We considered these CMPs because they all support, in some level, the Amazon EC2 service. Since the RDS service is quite new however, not all of them support it, but we will nonetheless evaluate their level of compatibility with it.

Furthermore, we also inquired further and selected a free IaaS provider, FutureGrid[3], as an example to deploy the given structure.

5.3.0.1 Eucalyptus

Eucalyptus[4] started as a research project at the University of Santa Barbara as a bridge for the VGrADS[5] and LEAD [6] projects going on at the time. The project resulted from the need to have a private cloud that could connect the VGrADS local systems to the Amazon public cloud for a faster development while still having the benefits of the public cloud [Eucf]. This early decision to support the Amazon's API had a strong impact in the project's early development goals and it is one of the main marks in the project's philosophy: to provide the best possible integration with AWSs. To sustain this decision, Eucalyptus has the best compatibility with Amazon's API of all the other CMPs studied in this work. Currently it supports the Amazon's EC2, S3, EBS, AMI and Identity and Access Management (IAM) services [Euca]. It's persistent storage system, Walrus, is strongly similar to Amazon's S3 service as well while also aiming at a very decentralized approach [MG11a; ST10].

The remaining core components of Eucalyptus are the Cloud Controller, which is the entry point into the system and is responsible for exposing and managing the underlying virtualized resources; the Cluster Controller, that it is usually a machine connected to the Node Controller and the Cloud Controller and that manages virtual machine networks as well as the scheduling on various Node Controllers; Storage Controller, that provides

[3] FutureGrid - https://portal.futuregrid.org/
[4] http://www.eucalyptus.com/
[5] http://vgrads.rice.edu/
[6] https://portal.leadproject.org/gridsphere/gridsphere

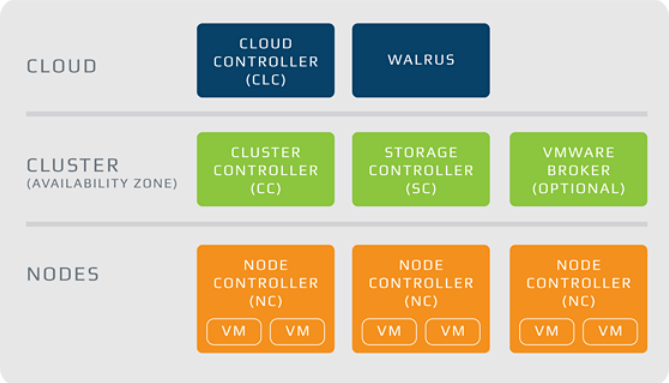

Figure 5.4: Main components of Eucalyptus from www.eucalyptus.com

the support for the Amazon EBS service; Node Controller, that executes on any machine hosting VMs controlling the VM activities there; and finally the optional VMware Broker, only available to Eucalyptus subscribers and that allows the deployment of VMs on VMware infrastructure elements [Eucd].

Eucalyptus is also built in Java, an open-source language back at the time of creation of the project, and thus its API naturally supports it. Besides Java, PHP is also supported and it can be found the project's GitHub account [Eucg]. As far as operating systems go, because Eucalyptus offers its source code to anyone, any potential customer can download it and build it [7]. However Eucalyptus offers support for CentOS, RHEL [Eucc] and there is a special Ubuntu version for it [Eucf]. In the field of hypervisors, the project has support for Xen, KVM [Euce] and VMware [Euch]. Although these hypervisors are supported, the level of performance between each host operating system and hypervisor will vary vastly and it is recommended that the users inform themselves with the community to choose the best pair for their needs.

Other important features of the project are its UI for users, which separates users and administrators by protecting users from low level details [ST10; MG11a] and its excellent support with one of the largest communities [MG11a; Pan13] and several support plans, for both enterprises who are willing to pay or for customers who simply wish to try the service [Eucb].

A tentative to install Eucalyptus was also done on two machines using the Intel VT technology, which resulted in a small report [8].

5.3.0.2 OpenStack

OpenStack was originally created by NASA and Rackspace with the main objective of producing an ubiquitous open source cloud computing platform for public and private clouds [SSCCBALDDP12; Opej].

Unlike other CMPs like Eucalyptus or OpenNebula, OpenStack does not take the

[7] https://github.com/eucalyptus/eucalyptus
[8] https://dl.dropboxusercontent.com/u/785815/Thesis_Extra/Intel.pdf

cloud-in-a-box approach. Instead OpenStack is better understood as a set of various tools and projects that allow for the construction of a cloud [Chu12; WGLGZ12; Quo]. The most important projects used in OpenStack are Compute (codename "Nova"), inherited from NASA and it acts as a main processing service; Object Storage (codename "Swift"), offered by Rackspace to fill the need of an object storage system; and the virtual image's manipulation system (codename "Glance"), which allows creation lookup and retrieval of virtual images [SSCCBALDDP12; WGLGZ12]. Additionally to these core projects, two more are important to mention, the dashboard (codename "Horizon") project, which provides a user interface based in the browser for an easier user experience and the identity service (codename "Keystone"), that provides a common authentication and authorization layer [WGLGZ12]. Other codenamed projects worth mentioning are Quantum [9] for managing networks and IP addresses; Cinder [10], that provides persistent block storage aiming to separate it from Nova; and Melange [11] that provides network information services across the platform.

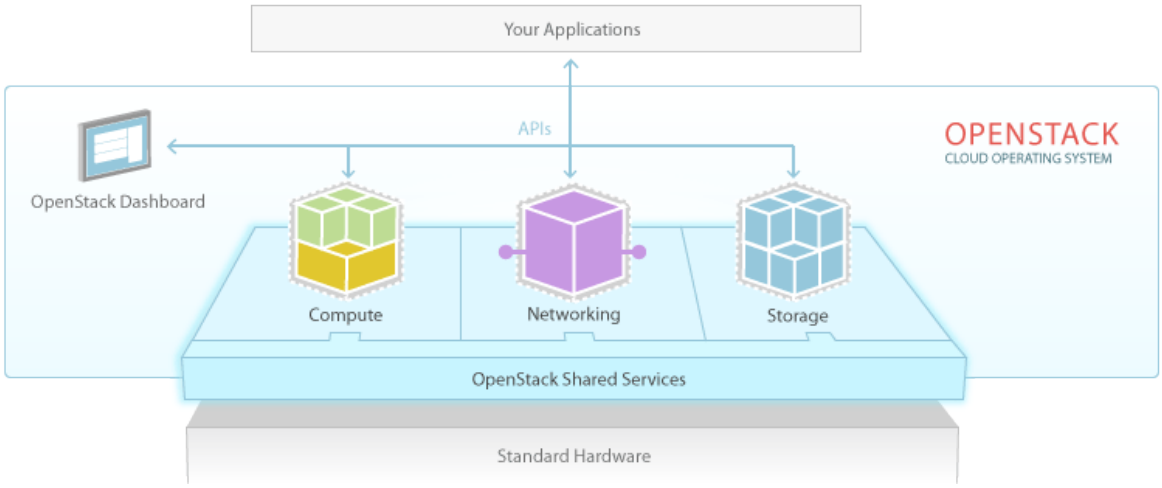

Figure 5.5: Main OpenStack services

This large set of projects supporting OpenStack is in great part due to the fact that at the time of this writing, OpenStack held the largest user community of all the CMPs considered here [Pan13] with more then eight thousand supporters in 87 countries and growing [Opej].

This large community is reflected not only in the amount of new core projects that are being born to keep up with Amazon Beta services (such as RDS [12]), but also in the support for host operating systems, programming languages and hypervisors. When compared to the other CMPs studied in this work, OpenStack supports more operating systems: Red Hat Enterprise Linux, Scientific Linux, CentOS, Debian, Ubuntu, openSuse, SLES

[9] https://wiki.openstack.org/wiki/Quantum#What_is_Quantum.3F
[10] https://wiki.openstack.org/wiki/Cinder#What_is_Cinder_.3F
[11] https://wiki.openstack.org/wiki/Melange
[12] http://aws.amazon.com/rds/

[Opea]; more programming languages: PHP, Java, Python, Ruby, C# [Opeb] and more hypervisors: Xen, KVM, HyperV, VMware, LXC [WGLGZ12]. Furthermore, if something is not supported, the users can download the source code [13]and make their own builds themselves.

As far as AWS support goes, however, OpenStack lacks many of the features present in Eucalyptus and some in OpenNebula. Even for Ec2, the support is incomplete and thrives on upgrades and side projects made by partners and the community, such as the AWSOME project by Canonical [Can]. Still, OpenStack is the only of all the studied CMPs that supports the Amazon's RDS service thanks to their new database as a service project codenamed "RedDwarf" [Spe11].

Overall OpenStack is a highly divided and customizable set of projects that interact in order to create highly scalable and distributed public and private clouds, full of many different features for everyone. This, however, has a negative side effect - with all this freedom many of the OpenStack versions and projects end up being widely different, like it happened to the UNIX universe, resulting in a scattered set of operative systems that although fill different niches, all fight among each other to get users and special interests in the market field [Dar12c].

5.3.0.3 OpenNebula

Established as a research project in 2005 but only released in March 2008, OpenNebula is by far the oldest of the studied CMPs. With Eucalyptus being started in 2007 [Eucf], OpenStack giving it's first steps in 2008 [**openstack_story**; Met12], CloudStack's prototype being developed in 2008 [Lia12] and Nimbus being released in 2009 [Nimd], OpenNebula had three years to grow by accumulating knowledge in the areas of management of virtual machines and large scale distribution of infrastructures, namely big data centers [WGLGZ12].

Consequently, OpenNebula's main focus is not to provide a low level IaaS platform like Eucalyptus [Ign13], but it is rather to virtualize large data centers into private clouds that share a small or medium sized set of trusted users [ST10]. The reason for this is the platform's architecture which is divided into three main layers: *Tools*, the outer layer that provides the command line interface, the browser and the libvirt API to allow communication between the user and the system; *the Core*, a centralized layer that manages the allocation of the dynamic IPs and image storage in disk for the virtual machines; and finally the *Drivers layer*, which communicates directly with the underlying operative system and encapsulates the platform into an abstract system.

It is this last layer that is responsible for creating, starting and stopping the virtual machines on every host independently of the hypervisor being used, while also monitoring their performance and the performance of the real machine [ST10; Win11], allowing the system to be fully restored if a crash should happen. Because this last layer is the door of

[13] https://launchpad.net/openstack/

communication between the host and the platform, it is also responsible for regulating the data transfers from every direction, including from external services such as the AWS [Win11].

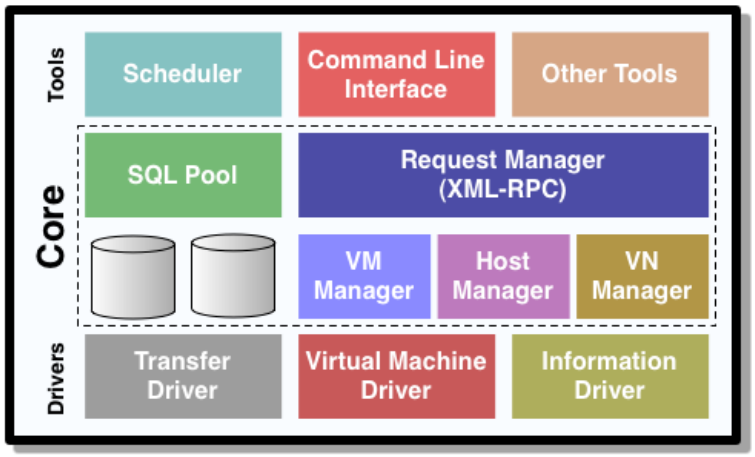

Figure 5.6: OpenNebula's Layers

While OpenStack and Eucalyptus have their own storage image systems like Walrus or Swift, OpenNebula uses a highly configurable centralized shared file system like Network File System (NFS), GlusterFS or SCP. While allowing live migration [ST10] and better speed when compared to the other two heavily distributed versions of its competitors, this approach is susceptible of becoming a bottleneck, and its high level of customization visible to the users allows for situations that are prone to serious errors. Furthermore, if the system is required to be secure, its setup totally is dependent on the administrator, requiring him to make all the necessary changes to the default NFS protocol or to replace it by another protocol like SCP [Win11].

Aside from its large number of possible customizations available to both administrator and user, OpenNebula also has official support for several operating systems, such as Debian, openSuse, Ubuntu, CentOS, RHEL, ArchLinux, Mac OSX, Fedora [Opeg; Oped; Opeh] with the source available on GitHub [14], and it also supports a set of well known programming languages: Java, C++, Ruby and Python, while offering code examples for each one [Opei; Opee].

This high customization tendency is also visible in the hypervisors supported by OpenNebula. Although at the time of writing OpenNebula only supports Xen, KVM, VMware and Libvirt with ESX, it plans on adding support for Hyper-V and VirtualBox as well [Vaz11; Tor12; Llo11; Tra], making it the platform with more choice in the field of which hypervisors to use.

The price for such customization however, comes in not having such a good support for the AWS. Although OpenNebula supports Amazon's EC2 service, it only supports the S3 service partially [Mon12; Opef]. In order to cope with this, other services are being

[14] https://github.com/OpenNebula/one

developed, such as the Amazon EBS service [Mon12] and OpenNebula also has support for third party tools [Opec] to facilitate the use of EC2.

Overall OpenNebula was built to be a customizable platform not bound to any specific type of environment [WGLGZ12] that is also easy and straightforward to install upon any system [Win11]. Having in mind that it has one of the smallest communities [Pan13] (excluding Nimbus), the set of customization possible is quite large. All this power, however, must be used correctly, thus requiring the users to have knowledge of what they are doing because they have access to the low levels of the platform. Therefore, OpenNebula is suited for researchers of computer science and universities who wish to experiment combining cloud systems with other technologies, or to companies with large data centers but with a small to medium set of trusted expert users.

5.3.0.4 Nimbus

Officially released in 2009 [Nimd], Nimbus advertises itself as a science cloud solution. Nimbus is affiliated with the Globus Project [15] and uses Globus credentials for the authentication of users, requiring them to be familiar with the x509 certificate, which is not common for non academic and scientific users. While still being highly customizable, Nimbus does a better job at protecting the user from the low level details when compared to OpenNebula leaving them for the administrator [ST10].

Nimbus main components are the Cumulus centralized storage system, which is considerably faster than Swift [PG13] and allows for system recovery through backups [Win11], a communication tier comprised by Nimbus-Web and a command line interface for interaction with the platform, and the network tier [Win11].

Unlike the other platforms where there is a specific network service, in Nimbus all that is necessary is that each node has a DHCP server in order to allow Nimbus to choose a random MAC address and SSH installed so processes can cleanly communicate between each other and between each nodes [ST10]. Although allowing for a broad set of virtual networks, this approach is not only less flexible, it also lacks elastic IP assignment [ST10; Tra].

Another important feature of Nimbus is the fact that it addresses the user's scheduling time as a primary concern that is directly integrated into the platform, unlike Eucalyptus or OpenNebula that use addons to measure it. Furthermore, research on allowing Amazon EC2 or other external cloud services to deal with excess demand is also being heavily investigates [ST10]. Nimbus also supports a wide variety of hypervisors and technologies to support and monitor virtual machines: Xen, KVM, Python, Bash, Ebtables, Libvirt. However, it lacks support for VMware [Win11], and it only supports two programming languages: Java and Pyhton [Win11; Nima]. Furthermore, documentation for supported host operating systems is very limited at best [Nimb]. Nimbus source code is also available in github [16]. As far as compatibility with AWSs goes, Nimbus partially supports EC2 and

[15] http://www.globus.org/

[16] Nimbus source - https://github.com/nimbusproject/nimbus

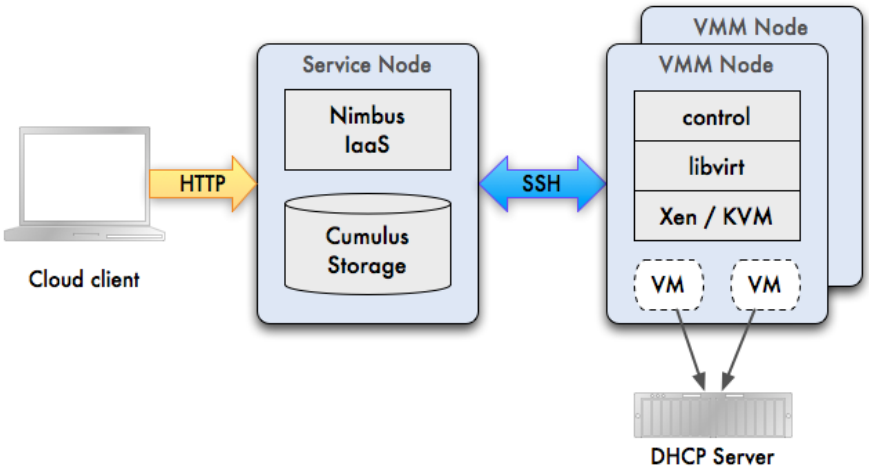

Figure 5.7: Nimbus's main components

supports S3 [Nimc; Win11], with no other known support projects being known publicly at the time of this writing.

Therefore, because of all the flexibility, knowledge required and the lack of good documentation when compared to the other CMPs here presented, Nimbus requires a high undertaking by its users and administrators [Tra; Win11]. With this in mind, Nimbus is therefore suited for scientific community clouds of universities, per example, where integration with other systems like Condor can be exploited allowing for the motorization of shared cluster time without have to deal with system specifications [ST10].

5.3.0.5 CloudStack

Although not officially supported by FutureGrid, CloudStack is still considered a major player in the open source CMP market, mainly thanks to its big community [Pan13]. Started in 2009 motivated by AWS success [Lia12] its main objective is to allow the creation of public, private, and hybrid clouds all alike for service providers and enterprises [Cloa; Cloc]. Still, despite this initial motivation, AWSs support is limited to EC2 and S3, with EC2 implementation being only partial. Future plans for EC2's support however are being planned [Cloj].

Unlike the other examples seen here, CloudStack's architecture is simple. It only has two parts: the management server, and the cloud infrastructure that it manages. The management server manages the system's resources, such as the hosts, storage devices, and IP addresses, while the infrastructure part is comprised of physical machines that have hypervisors ready to run VMs. It also deals with the allocation of VMs to certain hosts, the assignment of public and IP addresses, allocation of storage to guests, manages snapshots, templates and International Standards Organization (ISO) images, and abstracts the system to the other components [Cloc; Clod]. Furthermore, it also provides the users with a rich web user interface made using AJAX and built using HTML, CSS and jQuery [Cloc; Clob]. As far as the storage system used goes, CloudStack announces itself as using

NFS or Internet Small Computer System Interface (iSCSI), but it also support Swift from OpenStack, creating the SwiftStack project [17].

This versatility is also visible in the network capabilities of the platform. CloudStack has two types of networking scenarios: the basic scenario for an AWS-style network, and the advanced scenario for more sophisticated network topologies, which provides more flexibility in defining guest networks [Cloc].

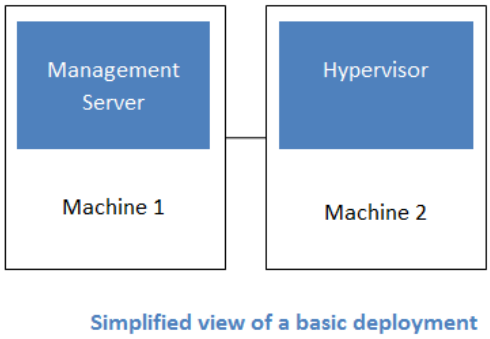

Figure 5.8: Simplified view of a CloudStack's deployment

CloudStack is also very flexible in the domain of languages it supports: although the main language is Java, there are also clients in Python, PHP and Perl [Clof; Clok; Clol; Cloi]. Furthermore, there is a client generator that can generate even more clients in more languages [Cloe]. In the domain of operating systems, it officially supports Debian, Red Hat, CentOS and Ubuntu [Cloc; Clog] and when it comes to hypervisors it supports XenServer, XCP, KVM, VMware [Cloh]. CloudStack's open source code can be found in github, and the Apache foundation provides a detailed project with the status of the platform and links to the source code, commiters, and other information [18].

Overall CloudStack is not as versatile as OpenStack, but its interaction with it makes it overall more flexible than Eucalyptus, while also trying to fit a different niche of the market by allowing more functionality than simple AWS support.

This platform is therefore well suited for large computer based companies with a powerful IT team, capable of managing all the details and tweaks of the network. As far as user trust-ability goes, the fact that it uses NFS as a preferential choice raises several problems already noted in the OpenNebula overview section. However those same users are shielded from some of the low level details thanks to the HTML5 interface provided by the management server. In this way, the users can be semi-trusted depending on the level of protection they get from the customizable web UI.

5.3.0.6 CMP Summary

There are many open source CMPs in the market and despite their differences most people think they are trying to reach for the same niche, and that in due time, there can only be

[17] http://swiftstack.com/cloudstack/
[18] http://incubator.apache.org/projects/cloudstack.html

one [Bro12; Ign13]. This however could not be farther from the truth - each project has its own philosophy, and based on it, a niche that it intends to occupy [Ign13].

Eucalyptus's main goal is to be as compatible as possible with AWS. Therefore it is more focused on infrastructure provisionement and is less flexible than a more virtual oriented approach such as OpenNebula, for example [Ign13].

OpenStack's philosophy is not to create a cloud in the box, but instead to provision a set of tools that together allow the administrators to build a flexible cloud. Thus, although both Eucalyptus and OpenStack aim at allowing the setup of private clouds, they do it very differently.

In the case of OpenNebula, they also aim at providing private clouds, but follow the model of heavy virtualization used in big data centers. This difference alone is so big that it allows a more liberate type of application development - in the Eucalyptus and OpenStack's case the applications must be built considering the infrastructure bellow them, while with OpenNebula that simply is not necessary [Ign13].

Nimbus is a special case since its main focus is to explore scientific research in community clouds. Although fitting a niche very close to infrastructure provision, but with special tools to deploy workloads, it can hardly be compared to the other CMPs studied. This is visible in its security protocol and partnership with the Globus project.

Finally, CloudStack aims to build public and private clouds for companies, and it has a very simple architectural system when compared to the other CMPs. CloudStack is actually quite close to OpenNebula in terms of flexibility and virtual management philosophy [Ign13], but it aims for a more profit oriented goal as stated in its documentation.

However, when choosing an open source CMP, its level of openness is also important as stated in [Llo13]. Although all projects have their code open to the community, there are other important aspects such as openness for the developer and for the customer. In the case of the developer, the level of openness defines how someone can contribute to the project, being the most different aspect the Governance Model. This model defines who chooses what can be done and what cannot be done in the project. Eucalyptus, OpenNebula and Nimbus (directed by the university of Chicago) all fall under the benevolent dictator category, while OpenStack is limited to the decision of a board of directors and CloudStack is bound to Apache's technical meritocracy [Llo13]. In terms of the customer lock in, Eucalyptus, Nimbus, OpenNebula and CloudStack all offer support for the Amazon's API, while OpenStack although supporting a set of Amazon's API is more focused in defining its own API and leaving its own mark on the territory.

As far as customer support is concerned, Eucalyptus, OpenNebula and CloudStack all have enterprise ready code. This happens because the community code is in fact the code that is going to be used by the enterprise, and if the customer desires additional support for the product he/she can buy it in addition.

This however is not the case for OpenStack, which only supports their customers if they buy enterprise specific versions of the platform, thus promoting irreversible lock in, not just to the platform being used, but also to the enterprise because OpenStack enterprise

versions are highly incompatible between each other.

Nimbus is an exception, because its main purpose is to support universities and scientific projects, enterprises usually do not chose it. Still, should such occur, Nimbus community version would have to be the same version used by the enterprise, but there would be no special support from Nimbus for it.

	Eucalyptus	OpenNebula	OpenStack	Nimbus	CloudStack
Philosophy	Integration with AWS	Private cutomizable cloud, through services and resource management	Public and Private clouds	Scientific research and communicty clouds	Public and private clouds that require a lot of scalability for service providers and enterprises
Ideal Settings	Large groups of machines and semi-trusted users	Large and small datacenters with highly trusted users that know what they are doing	Small or large group of machines with any range os users	Less to semi-trusted users with high level of knowledge	Semi-trusted Users with powerful IT team and large computer base
Programming Languages	Java, PHP	Java, C++, Ruby, Python	PHP, Java, Python, Ruby, C#	Java, Python	Java, PHP, Python, Perl
Cloud Types Supported	Private / Hybrid	Private / Public / Hybrid	Private / Public	Private / Community	Public / Private / Hybrid
Compatibility with AWS	EC2, S3, EBS, AMI, IAM	EC2, S3(subset)	EC2(partial, improved by AWSOME), S3, RDS	EC2(partial), S3	EC2(partial), S3
Suported Operative Systems	CentOS, RHEL	Debian, openSuse, Ubuntu, CentOS, RHEL, ArchLinux, Mac OSX, Fedora	RHEL, Scientific Linux, CentOS, Debian, Ubuntu, openSuse, SLES	Linux or Unix variant (OSX)	Debian, RHEL, CentOS, Ubuntu
Hypervisors Supported	Xen, KVM, VMware	Xen, KVM, VMware, Libvirt with ESX (VirtualBox and Hyper-V planned)	Xen, KVM, HyperV, VMware, LXC	Xen, KVM, Python, Bash, Ebtables, Libvirt	XenServer, XCP, KVM, VMware
Disk Image Storage	Walrus	Shared file system, like NFS, SCP or GlusterFS	Swift, Cinder	Cumulus	Preferably NFS and iSCSI, also compatible with Swift
User Interaction	euca2ools, Web tools	Interaction made using the Tools layer, user is exposed to low level details	Horizon	Command line, Nimbus Web	Customizable AJAX based Web UI

Table 5.1: CMP comparison table

6

Conclusions and Future Work

6.1 Conclusions

The existing middleware, previously developed by [Dom13], supports the concept of a session. This concept allows the middleware to aggregate heterogeneous data-sources and to disseminate the data received by them to a large number of clients that share a common interest.

However, this middleware was not without problems. The analysis of the architecture performed in Chapter 3 led us to believe that it would not scale if high frequency data sources were used, or if there were a large number of users in the sessions managed by the middleware. Following this hypotheses we then proceeded to inspect what was in fact inefficient, or even wrong, and what could be improved.

The first step of the analysis procedure was then to build a test platform. This test platform included the research and installation of an open-source XMPP server, in this case OpenFire, together with all their setup and configurations. We then created several tests in order to stress the middleware and observe if the performance problems were in fact important enough to deserve our attention. All this process required some effort from us since the mere task of running and evaluating the tests required changes to the middleware. With the tests ready we were finally able to write useful data in the logs and collect sound information from them about the middleware's behavior.

Along with the effort of studying and adapting the middleware, we also made an exhaustive research on which performance evaluator to use. This research led us to the conclusion that the Java visualVM was the most practical as well as the easiest evaluator to use. Not only did it not require further changes to the middleware's, it also came out of the box with the JVM and had a user friendly interface. Moreover, the Java visualVM

provided us with enough information for a raw analysis of the CPU usage and threads, which was our main interest.

Once the logs were done, the next step was the evaluation of the output generated via human eye (in order to identify of the most interesting parts) and via Java applications created for that specific purpose. One should have in mind that most logs had millions of lines, with millions of messages per run, and that we conducted over 250 tests. Although this task alone consumed a long period of time in the making of the thesis, it also allowed us to collect enough proof about the middleware's performance problems and to make an operational analysis which pin-pointed the exact location of the problems.

With the exact location of the problems discovered, two possibilities arose: were the performance issues in Esper or in the Camel routes? Once again, we had to run an exhaustive research to answer the above question and we concluded that although the usage of Esper in the context of the middleware could in fact be improved, there were no major gains from doing it. The major gain on improving the middleware's efficiency was in the Camel routes since these presented the main performance bottlenecks on data management due to their non-parallel nature.

With this new information we then created and explored two possible scenarios, one regarding multiple sessions, and the other regarding multiple clients (see Section 4.1). Our work mainly focused on scenario one, and after considering several options to increase the level of parallelization inside the same VM we finally ended up picking the SEDA option. After repeating the same battery of tests for the improved versions of the middleware and carefully making both a performance and operational analysis of the results, we concluded that the SEDA option had a positive impact in the middleware's performance, greatly reducing the execution time in all the tests, for all the Esper expressions used.

With the middleware running efficiently inside the same JVM by using all of its available cores, another problem was considered: scalability, for even though the middleware had a significant improvement in performance it was not yet scalable. To address this problem we studied the deployment of the middleware in the cloud, which gave us two main options - to deploy the middleware in a commercial could, like Amazon, and to use their services; or to deploy the middleware in a private/hybrid cloud - using the infrastructure of an IaaS provider - like FutureGrid together with an open-source CMP.

Due to the limited time to conclude this thesis, we chose to deploy the middleware in the Amazon since its earlier versions had already been deployed there and because doing so would relieve us from the work of having to install and configure the middleware to work within a new IaaS provider and a new CMP. The Amazon deployment provided us with further insight in a faster way, giving us valuable knowledge which can be reused on future deployments using a different CMP.

Once the middleware was deployed and running in the Amazon, the next step was to evaluate a proper scaling strategy. This included two steps - to find a way to monitor the current VM, and a strategy that would allow us to create or kill new virtual machines according to the necessities. To tackle the first challenge we chose SIGAR, which allowed

us to retrieve system information from the VM independently of the operating system being used. With this API we were able to monitor the CPU load every 30 seconds and take an adequate decision on what to do next.

To tackle the second problem, we considered using two strategies: master/slave and peer-to-peer. The first strategy is easier to implement and control, however it is a centralized model, prone to failures should the master die. The second strategy is more complex and harder to implement, but it will make the scalable component of the middleware more robust, mainly because there will be no hard dependencies between machines.

To summarize all the work done, we conclude this section with an objective view of what has been achieved. The thesis started with the hypothesis that the middleware had efficiency problems that could be solved. After analyzing the middleware's components and performing a series of tests in Chapter 3, we confirmed that initial hypothesis. Then we followed in Chapter 4 by suggesting a set of solutions, and we focused our attention in improving the scenario running multiple sessions, which wielded very positive results in the execution times of the middleware, thus achieving our main objective of improving the middleware's performance.

In Chapter 5 we explored an additional area of the middleware and focused on its scalability. Here we contemplated two possible strategies to scale the middleware into a variable number of VMs and extended the middleware with a simple version of the master and slave strategy. However, even though this solution was implemented, we did not have the chance of testing on a mass scale basis, so we cannot confirm if the solution will wield the expected results. We do know however, that for a small scenario it scales as expected.

6.2 Future Work

Even though much has been done, much is still left to do. When testing the middleware our main focus lied in scenarios comprising multiple sessions. This means that scenarios with dozens or even hundreds of data-sources and clients were not tested, thus opening a path for future work in this area.

Stressing the middleware with data-sources using the current test platform will require the creation dozens or even hundreds of accounts manually in the OpenFire server, and stressing the middleware with dozens or hundreds of clients will either require too many web browser clients, or to learn and master a new testing platform, like the previously mentioned Tsung.

Furthermore, since in the testing setup both clients and data-sources are currently all running in the same instance, increasing their will deprive the middleware from resources and probably affect its performance in a negative way. Thus, to fix this problem, the testing platform will have to be wider and involve more machines - at least two so the middleware may remain unaffected.

Also, when making our parallelization analysis we excluded some options because the amount of work required did not justify the end results - such are the cases of the Esper improvements and the Recipients List component with Camel. In the future, both these solutions may be further analyzed and compared with the chosen option of SEDA.

In regards to the cloud, there is the scalability strategy used - currently we have a proof of concept implementation of the master and slave strategy, and implementing a peer to peer algorithm may prove to be beneficial. Fortunately, the implementation is general enough as to allow an easy transition to such a strategy, should that happen.

Finally, the performance enhancement of the original middleware as proposed and implemented in this thesis, and suggested as future work, allows therefore the middleware's effective use in realistic application scenarios. This, in turn, may open the way to novel extensions to the session abstraction's capabilities in supporting dynamic data management.

Bibliography

[AABCHLMRRTXZ05] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. "The design of the borealis stream processing engine". In: *In CIDR*. 2005, pp. 277–289.

[AGM08] S. Abiteboul, O. Greenshpan, and T. Milo. "Modeling the mashup space". In: *Proceedings of the 10th ACM workshop on Web information and data management*. WIDM '08. Napa Valley, California, USA: ACM, 2008, pp. 87–94. ISBN: 978-1-60558-260-3. DOI: 10.1145/1458502.1458517. URL: http://doi.acm.org/10.1145/1458502.1458517.

[Amd67] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[Cloa] *Apache CloudStack FAQ*. URL: http://cloudstack.apache.org/cloudstack-faq.html.

[Clob] *Apache CloudStack Features*. URL: http://cloudstack.apache.org/software/features.html.

[AKS12] I. Astrova, A. Koschel, and M. Schaaf. "Automatic Scaling of Complex-Event Processing Applications in Eucalyptus". In: *Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering*. CSE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 22–29. ISBN: 978-0-7695-4914-9. DOI: 10.1109/ICCSE.2012.14. URL: http://dx.doi.org/10.1109/ICCSE.2012.14.

[Euca] *AWS and Eucalyptus*. URL: http://www.eucalyptus.com/aws-compatibility.

[BGP12] A. Baptista, M. C. Gomes, and H. Paulino. "Session-Based Dynamic Interaction Models for Stateful Web Services." In: *IESS*. Ed. by M. Snene. Vol. 103. Lecture Notes in Business Information Processing. Springer, 2012, pp. 29–43. ISBN: 978-3-642-28226-3. URL: http://dblp.uni-trier.de/db/conf/icexss/icexss2012.html#BaptistaGP12.

[Bas11] F. Basra. *DataNucleus All in One Persistance with JDO, JPA and REST API*. 2011. URL: http://www.javaplex.com/blog/datanucleus-persistance-jdo-jpa-nosql/.

[Ber11] E. Bernard. *Overview of Hibernate OGM*. 2011. URL: https://community.jboss.org/wiki/OverviewOfHibernateOGM.

[BG11] E. Bernard and S. Grinovero. *Hibernate OGM: JPA on Infinispan*. 2011. URL: http://www.redhat.com/summit/2011/presentations/jbossworld/whats_new/wednesday/bernard_w_420_jpa_in_hibernate.pdf.

[BT11] H. A. Bheda and C. S. Thaker. "Article: Virtualization Driven Mashup Container in Cloud Computing PaaS Model". In: *IJCA Proceedings on International Conference on Computer Communication and Networks CSI-COMNET-2011* comnet.1 (2011). Published by Foundation of Computer Science, New York, USA, pp. 9–14.

[Bro12] J. Brockmeier. *It's Not Highlander: There Can Be More Than One Open Source Cloud*. 2012. URL: http://readwrite.com/2012/04/05/its-not-highlander-there-can-b.

[Can] *Canonical's AWSOME bridges Amazon and OpenStack Clouds | Canonical*. URL: http://tinyurl.com/734dybj.

[Opea] *Chapter 1. Installing OpenStack Walk-through*. URL: http://docs.openstack.org/trunk/openstack-compute/install/yum/content/ch_installing-openstack-overview.html.

[CCJOSVW12] G. Chen, K. Chen, D. Jiang, B. C. Ooi, L. Shi, H. T. Vo, and S. Wu. "E3: an Elastic Execution Engine for Scalable Data Processing." In: *JIP* 20.1 (2012), pp. 65–76. URL: http://dblp.uni-trier.de/db/journals/jip/jip20.html#ChenCJOSVW12.

[Chu12] M. Chua. *Fielding common questions at your Eucalyptus talk*. 2012. URL: http://blog.melchua.com/2012/02/09/fielding-common-questions-at-your-eucalyptus-talk/.

[Cis] *Cisco and Partners to Build World's Largest Global Intercloud*. 2014. URL: http://newsroom.cisco.com/release/1373639.

[Eucb] *Cloud Support Services*. URL: http://www.eucalyptus.com/eucalyptus-cloud/support.

[Cloc] A. Cloudstack. *Apache CloudStack CloudStack API Developer ' s Guide*. URL: https://cloudstack.apache.org/docs/en-US/Apache_CloudStack/4.0.2/html/API_Developers_Guide/.

[Clod] *CloudStack architecture*. 2012. URL: http://www.slideshare.net/cloudstack/cloudstack-architecture.

[Cloe] *CloudStack Client generator*. URL: https://github.com/qpleple/cloudstack-client-generator.

[Clof] *CloudStack Coding Conventions*. URL: http://cloudstack.apache.org/develop/coding-conventions.html.

[Clog] *CloudStack download*. URL: http://cloudstack.apache.org/downloads.html.

[Cloh] *CloudStack Main*. URL: http://cloudstack.apache.org/.

[Cloi] *CloudStack perl client*. URL: https://github.com/jasonhancock/cloudstack-perl-client.

[CKR07] J. Curtin, R. J. Kauffman, and F. J. Riggins. "Making the 'MOST' out of RFID Technology: A Research Agenda for the Study of the Adoption, Usage and Impact of RFID". In: *Inf. Technol. and Management* 8.2 (June 2007), pp. 87–110. ISSN: 1385-951X. DOI: 10.1007/s10799-007-0010-1. URL: http://dx.doi.org/10.1007/s10799-007-0010-1.

[Dar12a] F. Darema. *Dynamic Data Driven Applications Systems (DDDAS)*. Tech. rep. NSF, 2012. URL: http://spruce.teragrid.org/workshop/talks/Darema.pdf.

[Dar12b] F. Darema. *From Big Data to New Capabilities*. Tech. rep. Air Force Research Laboratory, 2012. URL: http://www.cse.psu.edu/conferences/cb2012/Darema-CyberBridgesKeynoteNSF-June-2012-OverlaysExpanded.pdf.

[Dar12c] B. Darrow. *The dark side of OpenStack*. 2012. URL: http://gigaom.com/2012/09/28/the-dark-side-of-openstack/.

[Data] *DataNucleus*. URL: http://www.datanucleus.org/.

[Datb] *DataNucleus AccessPlatform 3.2 Checklist*. URL: http://www.datanucleus.org/products/accessplatform/.

[Datc] *DataNucleus Architecture*. URL: http://www.datanucleus.org/products/datanucleus/architecture.html.

[Datd] *Datastores*. URL: http://www.datanucleus.org/products/accessplatform_2_2/datastores_thirdparty.html.

[DG08] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[Nima] *Documentation - Nimbus*. URL: http://www.nimbusproject.org/docs/current/elclients.html.

[Nimb] *Documentation - Nimbus*. URL: http://www.nimbusproject.org/docs/2.2/admin/quickstart.html.

[Dom13] J. N. S. T. Domingos. "On the cloud deployment of a session abstraction for service/data aggregation". MA thesis. Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2013. URL: http://run.unl.pt/handle/10362/9923.

[Eucc] *Download Eucalyptus*. URL: http://www.eucalyptus.com/download/eucalyptus.

[Cloj] *EC2 API support in CloudStack*. URL: https://cwiki.apache.org/CLOUDSTACK/ec2-api-support-in-cloudstack.html.

[Ecla] Eclipse. *EclipseLink Project*. URL: http://projects.eclipse.org/projects/rt.eclipselink.

[Eclb] Eclipse. *EclipseLink/Examples/JPA/NoSQL*. URL: http://wiki.eclipse.org/EclipseLink/Examples/JPA/NoSQL\#NoSQL.

[Eclc] Eclipse. *EclipseLink/FAQ/NoSQL*. URL: http://wiki.eclipse.org/EclipseLink/FAQ/NoSQL.

[Ecld] Eclipse. *NoSQL Platform Concepts*. URL: http://www.eclipse.org/eclipselink/documentation/2.5/concepts/nosql001.htm\#BJEIHEJG.

[Ecle] Eclipselink. *EclipseLink Documentation*. URL: http://www.eclipse.org/eclipselink/documentation/.

[Eclf] Eclipselink. *Non-SQL Standard Database Support: NoSQL*. URL: http://www.eclipse.org/eclipselink/documentation/2.5/concepts/app_tl_ext003.htm\#CJAECHBD.

[Ert94] W. Ertel. "On the definition of speedup". In: *PARLE'94 Parallel Architectures and Languages Europe*. Ed. by C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis. Vol. 817. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 289–300. ISBN: 978-3-540-58184-0. DOI: 10.1007/3-540-58184-7_109. URL: http://dx.doi.org/10.1007/3-540-58184-7_109.

[Eucd] *Eucalyptus Components*. URL: http://www.eucalyptus.com/docs/3.1/ig/euca_components.html.

[Esp] *Event Processing with Esper and NEsper*. 2014. URL: http://esper.codehaus.org/.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. "Cloud Computing and Grid Computing 360-Degree Compared". In: *Grid Computing Environments Workshop, 2008. GCE '08*. 2008, pp. 1–10. DOI: 10.1109/GCE.2008.4738445.

[GGL09] R. Geambasu, S. D. Gribble, and H. M. Levy. "CloudViews: Communal Data Sharing in Public Clouds". In: *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*. HotCloud'09. San Diego, California: USENIX Association, 2009. URL: http://dl.acm.org/citation.cfm?id=1855533.1855547.

[GK03] N. S. Good and A. Krekelberg. "Usability and Privacy: A Study of Kazaa P2P File-sharing". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '03. Ft. Lauderdale, Florida, USA: ACM, 2003, pp. 137–144. ISBN: 1-58113-630-7. DOI: 10.1145/642611.642636. URL: http://doi.acm.org/10.1145/642611.642636.

[Gri12] S. Grinovero. *Using JPA applications in the era of NoSQL: Introducing Hibernate OGM*. 2012. URL: http://www.slideshare.net/ptjug/using-jpa-applications-in-the-era-of-nosql-introducing-hibernate-ogm.

[GDJ06] S. Guha, N. Daswani, and R. Jain. *An Experimental Study of the Skype Peer-to-Peer VoIP System*. 2006. URL: http://saikat.guha.cc/pub/iptps06-skype/.

[GJPPMSV12] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. "StreamCloud: An Elastic and Scalable Data Streaming System". In: *IEEE Trans. Parallel Distrib. Syst.* 23.12 (Dec. 2012), pp. 2351–2365. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.24. URL: http://dx.doi.org/10.1109/TPDS.2012.24.

[Gus88] J. L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415.

[Har] G. Harrison. *10 things you should know about NoSQL databases*. URL: http://www.techrepublic.com/blog/10-things/10-things-you-should-know-about-nosql-databases.

[Hiba] Hibernate. *Hibernate commercial support*. URL: http://www.hibernate.org/community.

[Hibb] Hibernate. *Hibernate OGM Docs*. URL: http://docs.jboss.org/hibernate/ogm/4.0/reference/en-US/html/preface.html\#d0e100.

[Hibc] HibernateOGM. *Hibernate OGM (Object/Grid Mapper)*. URL: http://www.hibernate.org/subprojects/ogm.html.

[Hig11] R. Hightower. *Hibernate Object Mapping for NoSQL Datastores*. 2011. URL: http://www.infoq.com/news/2011/07/hibernateogm.

[Hil04] M. D. Hill. *What is Scalability?* Tech. rep. University of Wisconsin, Computer Sciences Department, 2004. URL: ftp://ftp.cs.wisc.edu/markhill/Papers/can90_scalability.pdf.

[IA11] C. Ibsen and J. Anstey. *Camel in Action*. 2011. URL: http://tinyurl.com/l6sbgjt.

[Ign13] Ignacio M. Llorente. *EUCALYPTUS, CLOUDSTACK, OPENSTACK AND OPENNEBULA: A TALE OF TWO CLOUD MODELS*. 2013. URL: http://blog.opennebula.org/?p=4042.

[Euce] *Install Hypervisors*. URL: http://www.eucalyptus.com/docs/3.1/ig/installing_hypervisors.html.

[IBNW09] C. Ireland, D. Bowers, M. Newton, and K. Waugh. "A Classification of Object-Relational Impedance Mismatch". In: *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*. 2009, pp. 36–43. DOI: 10.1109/DBKDA.2009.11. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5071809.

[IBYBF07] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[Clok] *jasonhancock/cloudstack-python-client · GitHub*. URL: https://github.com/jasonhancock/cloudstack-python-client.

[Jon12] B. Jones. *CERN: Big Science Meets Big Data*. Tech. rep. European Organization for Nuclear Research, 2012. URL: http://openlab.web.cern.ch/sites/openlab.web.cern.ch/files/presentations/BobJones-BigDataAnalytics-Dec2012.pdf.

[Kar10] M. Karabisti. *Eclipse Link & Oracle Top Link ?* 2010. URL: http://www.coderanch.com/t/571557/ORM/databases/Eclipse-Link-Oracle-Top-Link.

[KBMBCDGFKMSSS02] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic. "People, Places, Things: Web Presence for the Real World". In: *Mob. Netw. Appl.* 7.5 (Oct. 2002), pp. 365–376. ISSN: 1383-469X. DOI: 10.1023/A:1016591616731. URL: http://dx.doi.org/10.1023/A:1016591616731.

[KTP09] A. Koschmider, V. Torres, and V. Pelechano. "Elucidating the Mashup Hype: Definition, Challenges, Methodical Guide and Tools for Mashups". In: *2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*. Madrid, Spain, 2009.

[Opeb] *Language-Specific API Bindings*. URL: http://docs.openstack.org/trunk/openstack-object-storage/admin/content/language-specific-api-bindings.html.

[Lia12] S. Liang. *The CloudStack development story and future vision*. 2012. URL: http://fr.slideshare.net/kkitase/cloudstackdevelopmen

[Llo11] I. M. Llorente. *OpenNebula Adds Hyper-V Support*. 2011. URL: http://blog.opennebula.org/?p=1991.

[Llo13] I. M. Llorente. *OPENSTACK, CLOUDSTACK, EUCALYPTUS AND OPENNEBULA: WHICH CLOUD PLATFORM IS THE MOST OPEN?* 2013. URL: http://blog.opennebula.org/?p=4372.

[MG11a] C. M. and P. G. *Open-Source Cloud Platforms*. 2011. URL: https://docs.google.com/presentation/d/1It2Kv594Mx0kZ9qgNA1i_PiKMY9HyrrvSMnm7XzCWLc/present\#slide=id.i0.

[MCBBDRB11] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. *Big Data: The Next Frontier for Innovation, Competition, and Productivity*. Tech. rep. McKinsey Global Institute, 2011.

[MC11] A. Margara and G. Cugola. "Processing flows of information: from data stream to complex event processing". In: *Proceedings of the 5th ACM international conference on Distributed event-based system*. DEBS '11. New York, New York, USA: ACM, 2011, pp. 359–360. ISBN: 978-1-4503-0423-8. DOI: 10.1145/2002259.2002307. URL: http://doi.acm.org/10.1145/2002259.2002307.

[MG11b] P. M. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011.

[Met12] C. Metz. *The Secret History of OpenStack, the Free Cloud Software That's Changing Everything*. 2012. URL: http://www.wired.com/wiredenterprise/?p=14392.

[Mew12] C. Mewawalla. *Big Data*. Tech. rep. CM Research, 2012. URL: http://www.cmresearch.co.uk/resources/Sync+43+Big+Data.pdf.

[MR06] S. Mirotic and I. Radusinovic. "The Principles of Speech Transmission Realization in Skype". In: *Proceedings of the 6th WSEAS International Conference on Multimedia, Internet & Video Technologies*. MIV'06. Lisbon, Portugal: World Scientific, Engineering Academy, and Society (WSEAS), 2006, pp. 1–6. ISBN: 960-8457-53-X. URL: http://dl.acm.org/citation.cfm?id=1365598.1365599.

[mon] mongodb. *NoSQL Databases Explained*. URL: http://www.mongodb.com/learn/nosql.

[Mon12] R. S. Montero. *Question about OpenNebula / Amazon S3*. 2012. URL: http://lists.opennebula.org/pipermail/users-opennebula.org/2012-August/020070.html.

[MCBFR08] S. Mosser, F. Chauvel, M. Blay-Fornarino, and M. Riveill. "Web Services Composition: Mashups Driven Orchestration Definition". In: *Proceedings of the 2008 International Conference on Computational Intelligence for Modelling Control & Automation*.

CIMCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 284–289. ISBN: 978-0-7695-3514-2. DOI: 10.1109/CIMCA.2008.96. URL: http://dx.doi.org/10.1109/CIMCA.2008.96.

[Myy12] Myy. *What exactly is Apache Camel?* 2012. URL: http://stackoverflow.com/questions/8845186/what-exactly-is-apache-camel.

[Nimc] *Nimbus Frequently Asked Questions — Nimbus 1.0 documentation*. URL: http://www.nimbusproject.org/doc/nimbus/faq/\#what-ec2-operations-are-supported.

[Nimd] *Nimbus information Wiki*. URL: http://tinyurl.com/ygv3665.

[Opec] OpenNebula. *EC2 Ecosystem*. URL: http://opennebula.org/documentation:archives:rel2.0:ec2qec.

[Oped] *OpenNebula - Open Source Data Center Virtualization*. URL: http://opennebula.org/documentation:rel3.8:ignc\#archlinux_platform_notes.

[Opee] *OpenNebula - OpenNebula Development pages*. URL: http://dev.opennebula.org/projects/opennebula/wiki/GSOC2011-StudentProjectIdeas/diff/18.

[Opef] *OpenNebula 1.4 RC*. 2009. URL: http://opennebula.org/software:rnotes:rn-rel1.4c.

[Opeg] *OpenNebula Download*. URL: http://opennebula.org/software:software.

[Opeh] *OpenNebula Platform Notes 2.0*. URL: http://opennebula.org/documentation:archives:rel2.0:notes.

[Opei] *OpenNebula Programming Examples*. URL: http://opennebula.org/documentation:archives:rel1.2:api_examples.

[Opej] *OpenStack: The 5-minute Overview*. URL: http://www.openstack.org/.

[Ora07] Oracle. *ECLIPSE PERSISTENCE PLATFORM (ECLIPSELINK) FAQ*. 2007. URL: http://web.archive.org/web/20070311212527/http://www.oracle.com/technology/tech/eclipse/pdf/eclipselink-faq.pdf.

[Pac11] P. Pacheco. *An Introduction to Parallel Programming*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 9780123742605.

[Pan13] J. Panettieri. *OpenStack vs CloudStack: The Latest Score*. 2013. URL: http://talkincloud.com/cloud-computing-management/openstack-vs-cloudstack-latest-score.

[PG13] A. Paul and K. Gill. *OpenStack-Swift Vs Nimbus-Cumulus*. Tech. rep. UCSB, 2013. URL: http://cs.ucsb.edu/~arghyadip/Project_Report.pdf.

[PRC13] B. Peer, P. Rajbhoj, and N. Chathanur. *Complex Events Processing: Unburdening Big Data Complexities*. Tech. rep. Infosys Labs, 2013. URL: http://www.infosys.com/infosys-labs/publications/Documents/complex-events-processing.pdf.

[Date] *Possible DataNucleus Future Plans*. URL: http://www.datanucleus.org/servlet/wiki/display/ENG/Release+and+Plans.

[Clol] *qpleple/cloudstack-php-client · GitHub*. URL: https://github.com/qpleple/cloudstack-php-client.

[Datf] *RDBMS Adapters*. URL: http://www.datanucleus.org/extensions/datastore_adapter.html.

[Res07] L. Resende. "Handling heterogeneous data sources in a SOA environment with service data objects (SDO)". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 895–897. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247582. URL: http://doi.acm.org/10.1145/1247480.1247582.

[RB05] M. Rouse and E. Blair. *throughput*. 2005. URL: http://searchnetworking.techtarget.com/definition/throughput.

[RB06] M. Rouse and E. Blair. *latency*. 2006. URL: http://whatis.techtarget.com/definition/latency.

[sal12] salesforce. *Why Move to the Cloud? 10 Benefits of Cloud Computing*. 2012. URL: http://www.salesforce.com/uk/socialsuccess/cloud-computing/why-move-to-cloud-10-benefits-cloud-computing.jsp.

[SM12] A. Salminen and T. Mikkonen. "Mashups - Software Ecosystems for the Web Era." In: *IWSECO@ICSOB*. Ed. by S. Jansen, J. Bosch, and C. F. Alves. Vol. 879. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 18–32. URL: http://dblp.uni-trier.de/db/conf/icsob/iwseco2012.html#SalminenM12.

[Sem] A. Semenov. *Evolution of Peer-to-peer algorithms: Past, present and future.* Tech. rep. Helsinki University of Technology. URL: http://www.tml.tkk.fi/Publications/C/18/semenov.pdf.

[ST10] P. Sempolinski and D. Thain. "A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus". In: *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 417–426. ISBN: 978-0-7695-4302-4. DOI: 10.1109/CloudCom.2010.42. URL: http://dx.doi.org/10.1109/CloudCom.2010.42.

[Ske11] J. H. Skeie. *Effective Java Profiling With Open Source Tools*. 2011. URL: http://www.infoq.com/articles/java-profiling-with-open-source.

[SSCCBALDDP12] D. Souza, T. Sena, E. Cavalcante, N. Cacho, T. Batista, A. Almeida, F. Lopes, T. Diniz, F. C. Delicato, and P. F. Pires. *Implantação de Aplicações em Múltiplas Plataformas de Nuvem*. Tech. rep. Dimap, Universidade Federal do Rio Grande do Norte, 2012. URL: http://www.dimap.ufrn.br/~altostratus/downloads/publicacoes/2012_WCGA-Souza.pdf.

[Spe11] S. Spector. *Announcing Project RedDwarf – Database as a Service*. 2011. URL: http://www.openstack.org/blog/2011/04/announcing-project-reddwarf-database-as-a-service/.

[SFDM12] M. Stecca, M. Fornasa, N. Dall'Armellina, and M. Maresca. "Event-Driven Mashup Orchestration with Scala". In: *Proceedings of the 2012 IEEE Ninth International Conference on Services Computing*. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 531–538. ISBN: 978-0-7695-4753-4. DOI: 10.1109/SCC.2012.41. URL: http://dx.doi.org/10.1109/SCC.2012.41.

[SM10] M. Stecca and M. Maresca. "An Architecture for a Mashup Container in Virtualized Environments". In: *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 386–393. ISBN: 978-0-7695-4130-3. DOI: 10.1109/CLOUD.2010.34. URL: http://dx.doi.org/10.1109/CLOUD.2010.34.

[SMKKB01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA: ACM, 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: 10.1145/383059.383071. URL: http://doi.acm.org/10.1145/383059.383071.

[Eucf] *The Eucalyptus Story*. URL: http://www.eucalyptus.com/about/story.

[Web] *The webapp2 Framework*. 2014. URL: https://developers.google.com/appengine/docs/python/tools/webapp2.

[Thr] *Throughput*. URL: http://en.wikipedia.org/wiki/Throughput.

[Tor12] G. Toraldo. *Supporting hypervisors by OpenNebula*. 2012. URL: http://www.packtpub.com/article/supporting-hypervisors-opennebula.

[Tra] V. Tran. *Introduction to Cloud computing*. URL: http://indico.egi.eu/indico/getFile.py/access?contribId=8\&resId=0\&materialId=slides\&confId=183.

[Eucg] *Using PHP with Eucalyptus*. URL: https://github.com/eucalyptus/eucalyptus/wiki/Using-PHP-with-Eucalyptus.

[VRMCL08] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. "A Break in the Clouds: Towards a Cloud Definition". In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 2008), pp. 50–55. ISSN: 0146-4833. DOI: 10.1145/1496091.1496100. URL: http://doi.acm.org/10.1145/1496091.1496100.

[Var95] G. Varghese. *CS243 Computer Communications: More Error Recovery*. Tech. rep. Washington University, Department of Computer Science, 1995. URL: http://cseweb.ucsd.edu/users/varghese/TEACH/cs123/swindow.pdf.

[Vaz11] T. Vazquez. *Hypervisors supported by Open Nebula 2.2*. 2011. URL: http://lists.opennebula.org/pipermail/users-opennebula.org/2011-July/005851.html.

[Euch] *VMware Support*. URL: http://www.eucalyptus.com/docs/3.1/ig/planning_vmware.html.

[Wah11] K. Wahner. *When to use Apache Camel?* 2011. URL: http://www.kai-waehner.de/blog/2011/06/02/when-to-use-apache-camel/.

[WGLGZ12] X. Wen, G. Gu, Q. Li, Y. Gao, and X.-J. Zhang. "Comparison of open-source cloud management platforms: OpenStack and OpenNebula." In: *FSKD*. IEEE, 2012, pp. 2457–2461. URL: http://dblp.uni-trier.de/db/conf/fskd/fskd2012.html#WenGLGZ12.

[Quo] *What are the advantages and the disadvantages of OpenStack vs. Eucalyptus, and enStratus vs. Scalr vs. RightScale?* 2012. URL: http://www.quora.com/What-are-the-advantages-and-the-disadvantages-of-OpenStack-vs-Eucalyptus-and-enStratus-vs-Scalr-vs-RightScale.

[Datg] *Why DataNucleus*. URL: http://www.datanucleus.org/project/why.html.

[Win11] S. Wind. "Open source cloud computing management platforms: Introduction, comparison, and recommendations for implementation". In: *Open Systems (ICOS), 2011 IEEE Conference on*. Sept. 2011, pp. 175–179.

[WDR06] E. Wu, Y. Diao, and S. Rizvi. "High-performance complex event processing over streams". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 407–418. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142520. URL: http://doi.acm.org/10.1145/1142473.1142520.

[XGS11] L. Xiong, S. Goryczka, and V. Sunderam. "Adaptive, Secure, and Scalable Distributed Data Outsourcing: A Vision Paper". In: *Proceedings of the 2011 Workshop on Dynamic Distributed Data-intensive Applications, Programming Abstractions, and Systems*. 3DAPAS '11. San Jose, California, USA: ACM, 2011, pp. 1–6. ISBN: 978-1-4503-0705-5. DOI: 10.1145/1996010.1996012. URL: http://doi.acm.org/10.1145/1996010.1996012.

[ZCB10] Q. Zhang, L. Cheng, and R. Boutaba. "Cloud computing: state-of-the-art and research challenges". English. In: *Journal of Internet Services and Applications* 1.1 (2010), pp. 7–18. ISSN: 1867-4828. DOI: 10.1007/s13174-010-0007-6. URL: http://dx.doi.org/10.1007/s13174-010-0007-6.

Performance Information

A.1 1 source, 1 session, 1 client full info

A.1.1 visualVM monitor screenshots

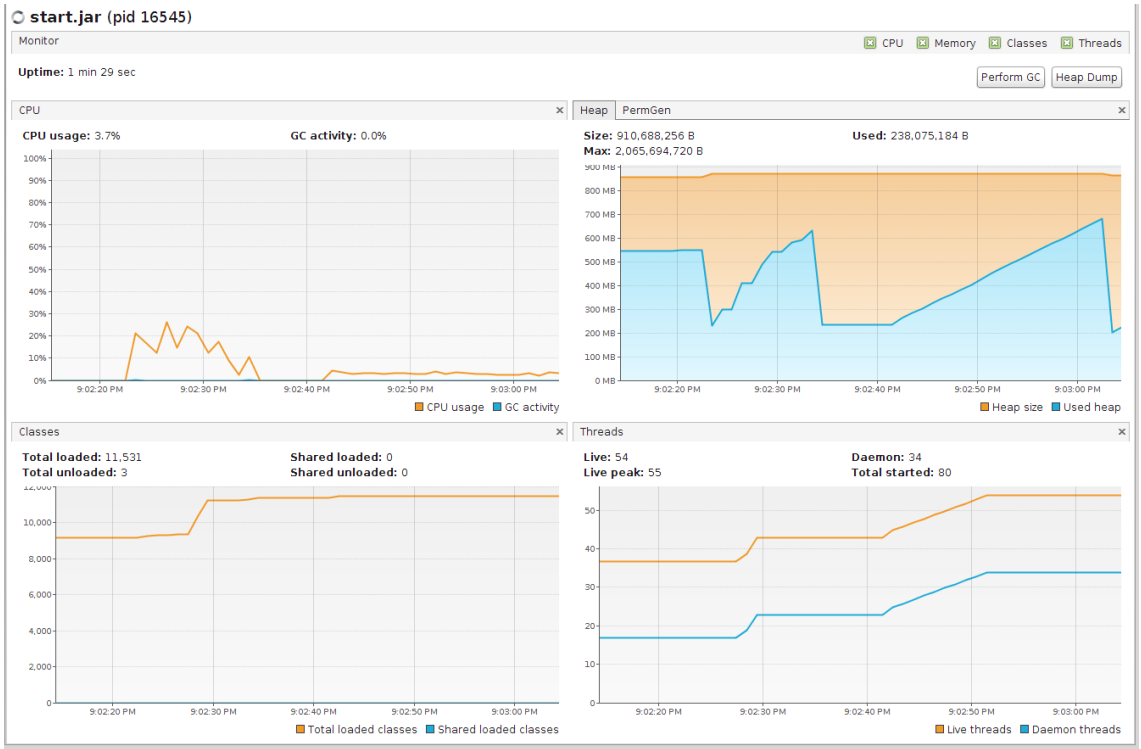

Figure A.1: Resources used when running Exp0 with 60 messages per minute

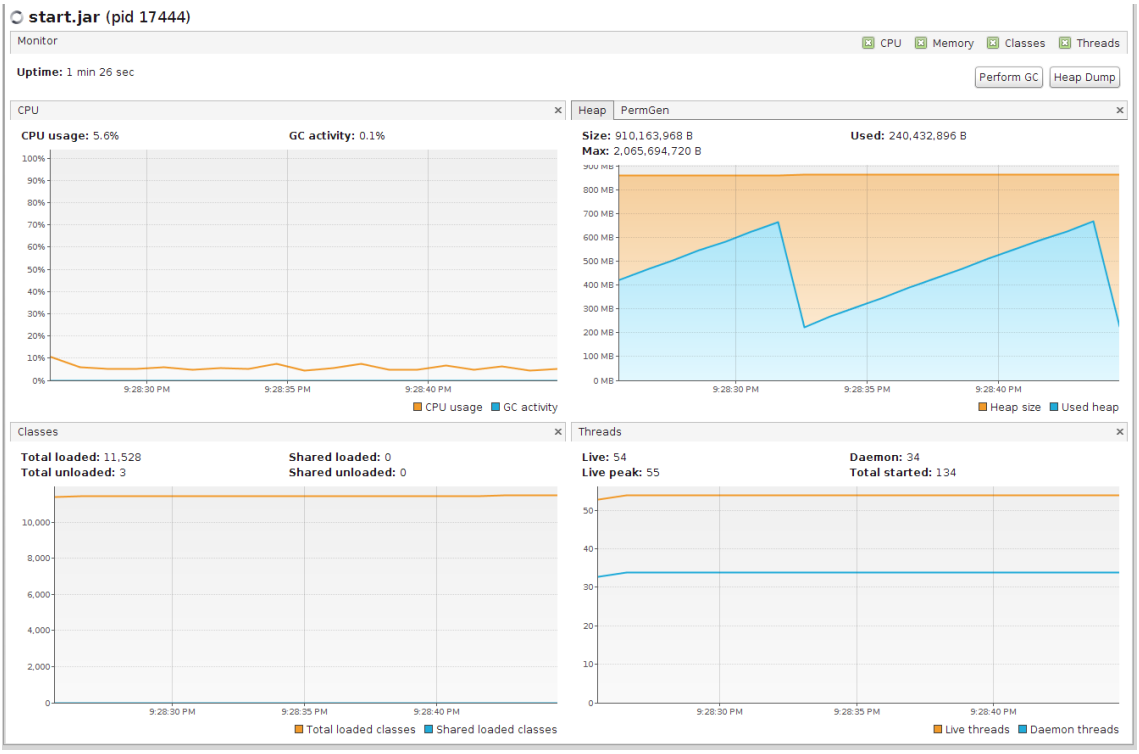

Figure A.2: Resources used when running Exp0 with 120 messages per minute

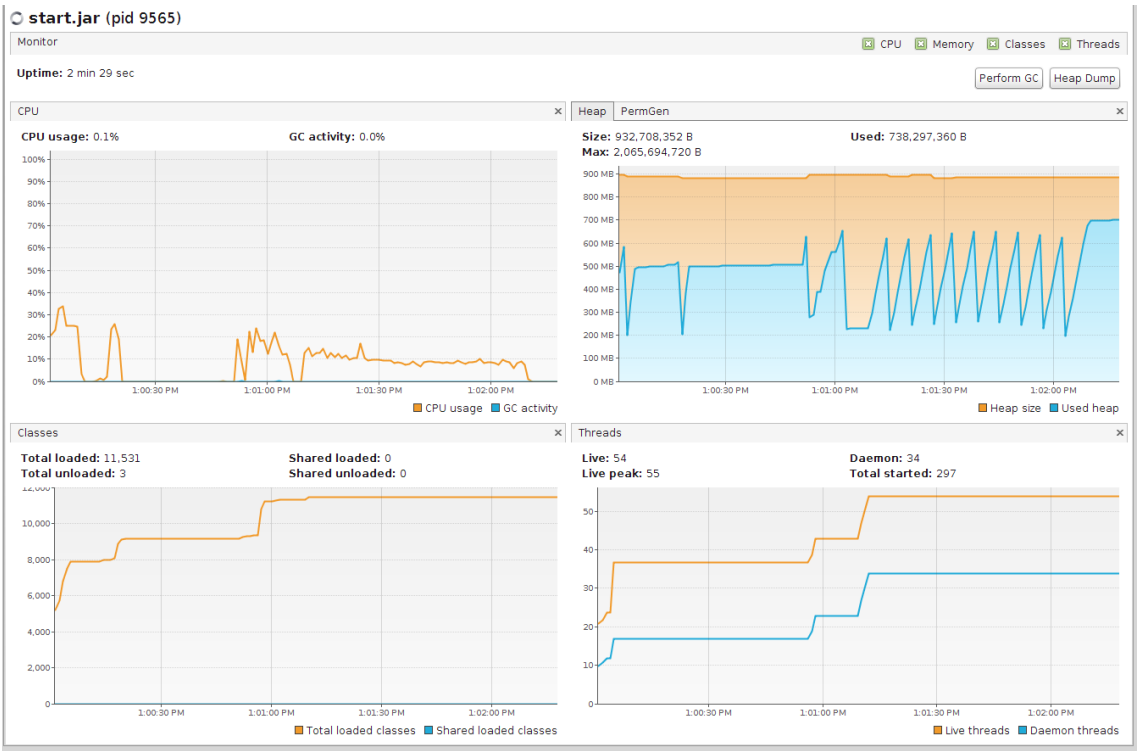

Figure A.3: Resources used when running Exp0 with 240 messages per minute

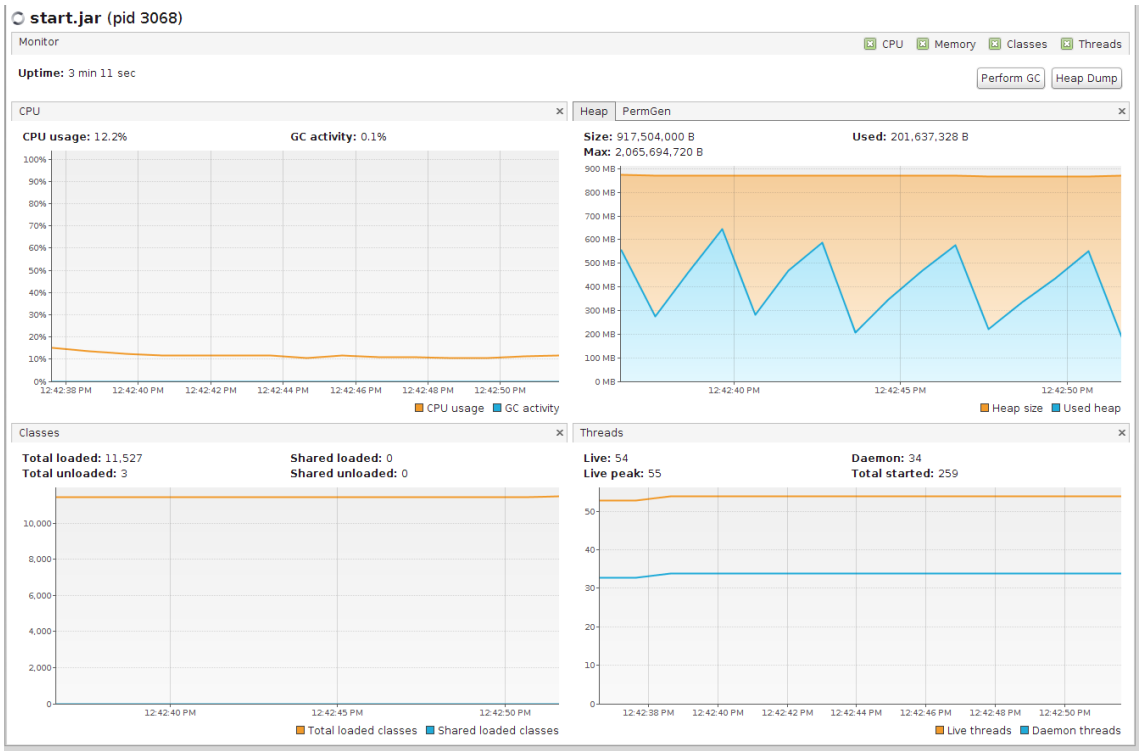

Figure A.4: Resources used when running Exp0 with 480 messages per minute

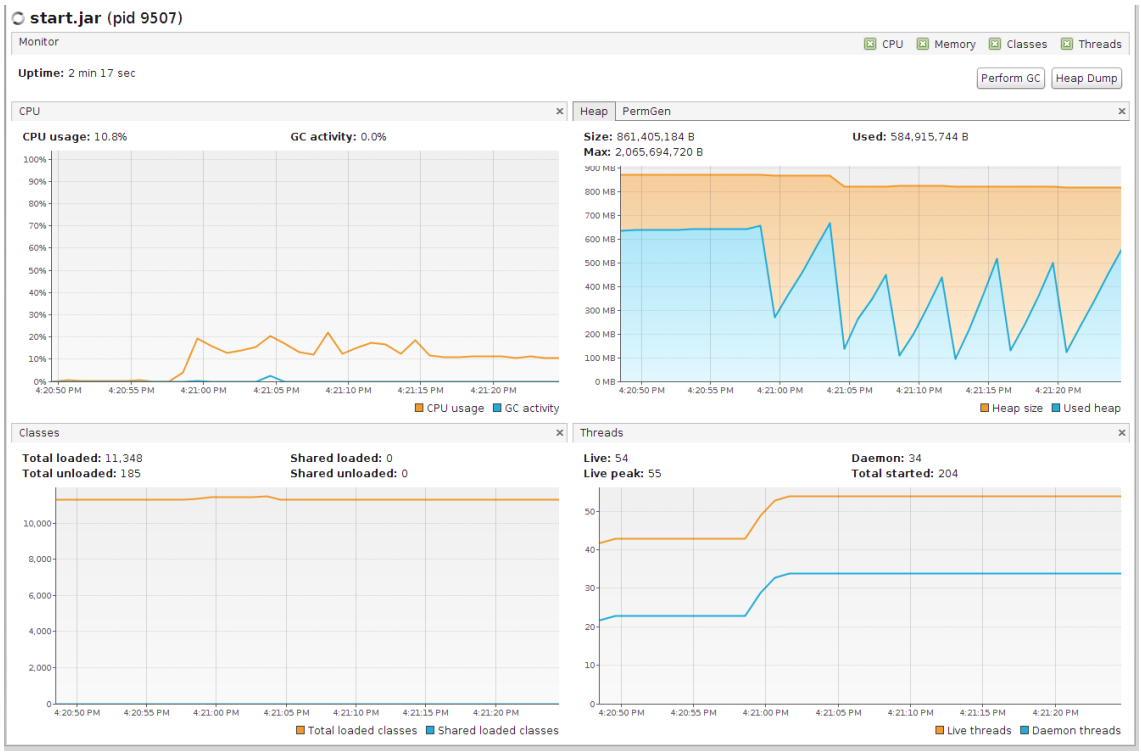

Figure A.5: Resources used when running Exp0 with 1000 messages per minute

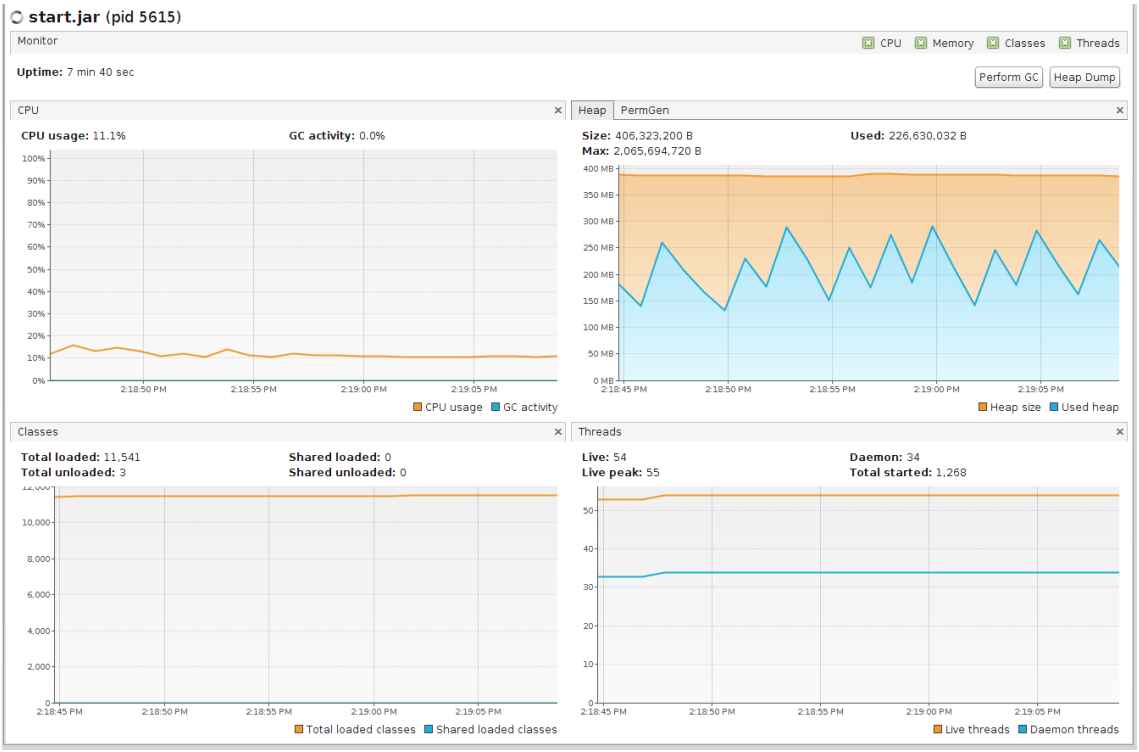

Figure A.6: Resources used when running Exp0 with 2000 messages per minute

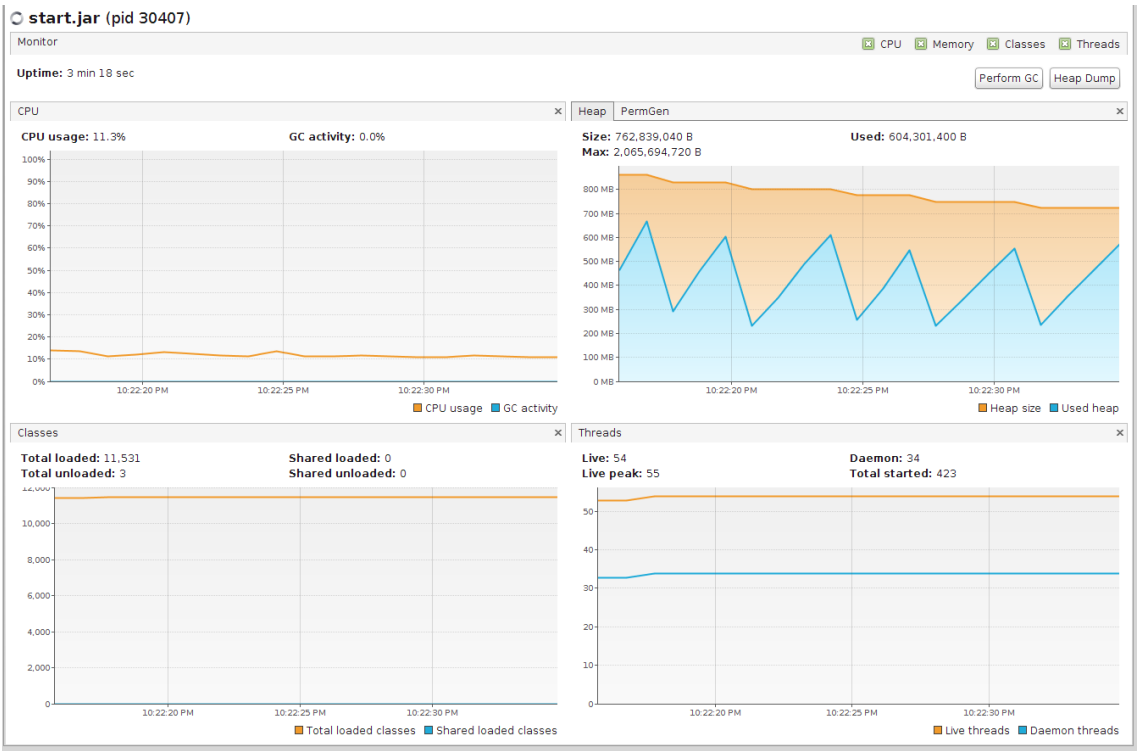

Figure A.7: Resources used when running Exp0 with 4000 messages per minute

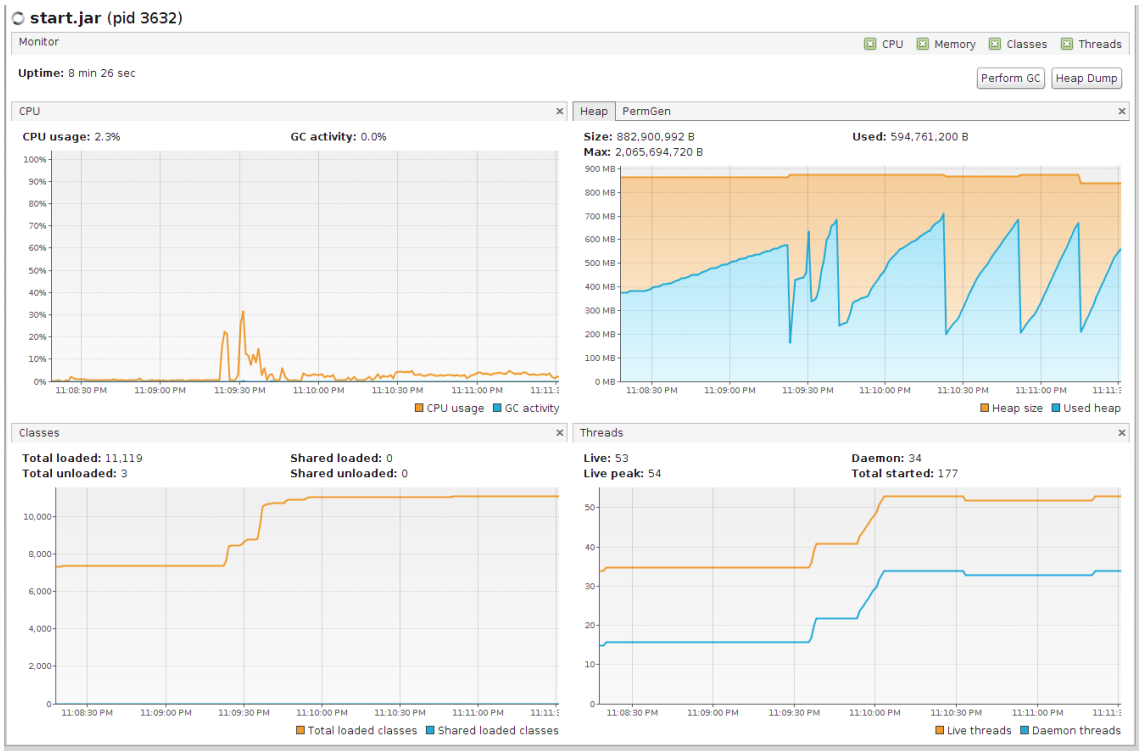

Figure A.8: Resources used when running Exp6 with 60 messages per minute

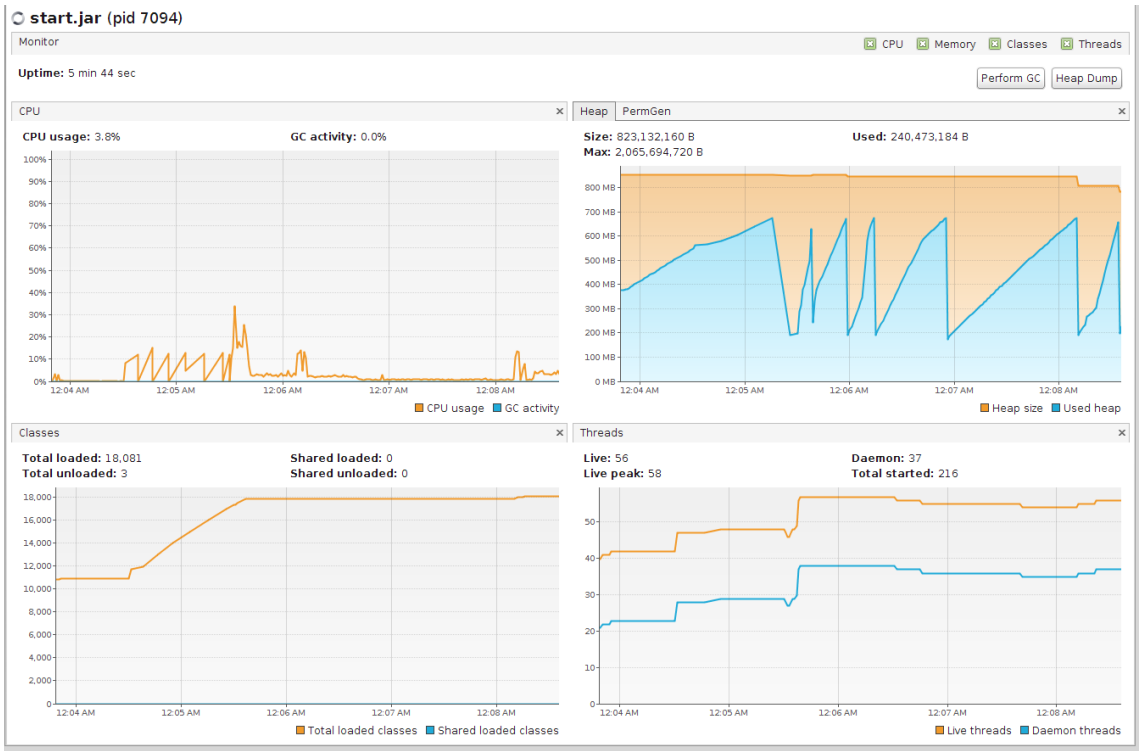

Figure A.9: Resources used when running Exp6 with 120 messages per minute

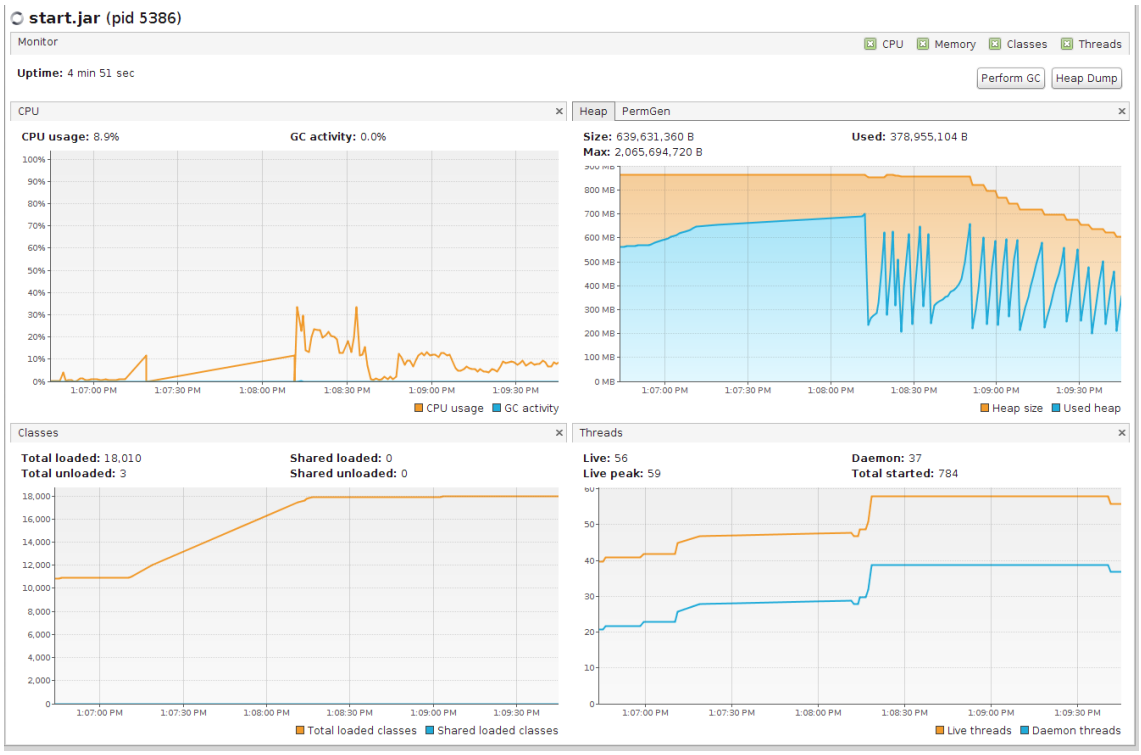

Figure A.10: Resources used when running Exp6 with 240 messages per minute

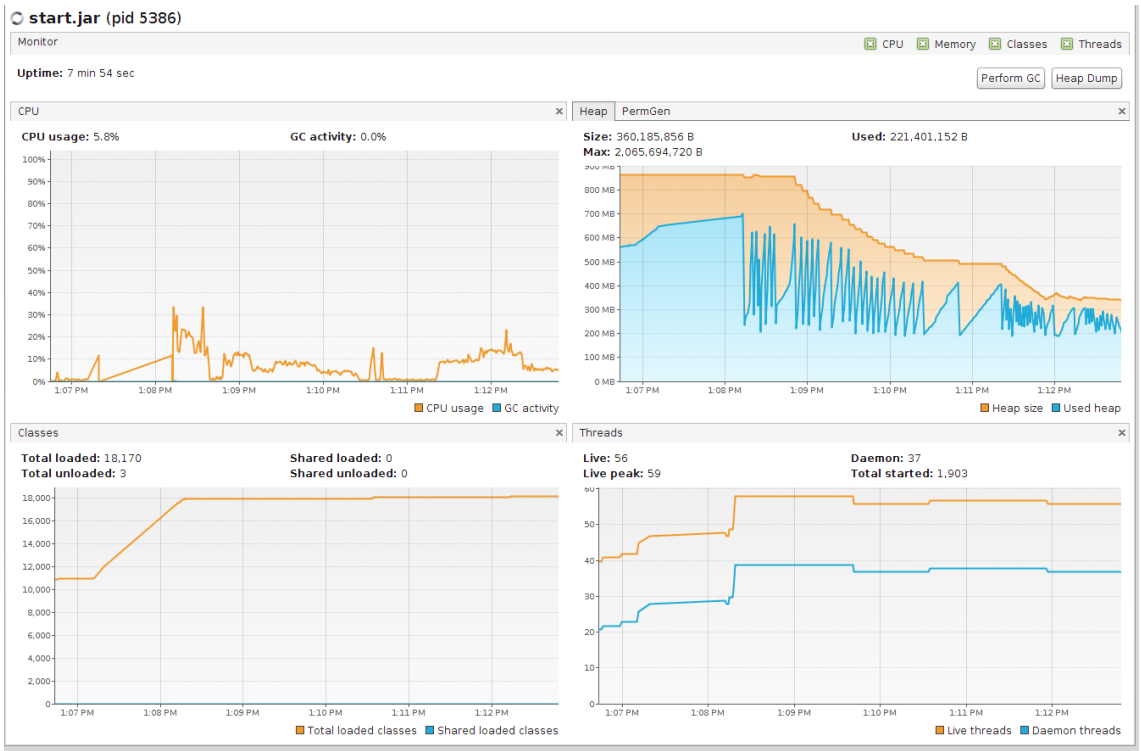

Figure A.11: Resources used when running Exp6 with 480 messages per minute

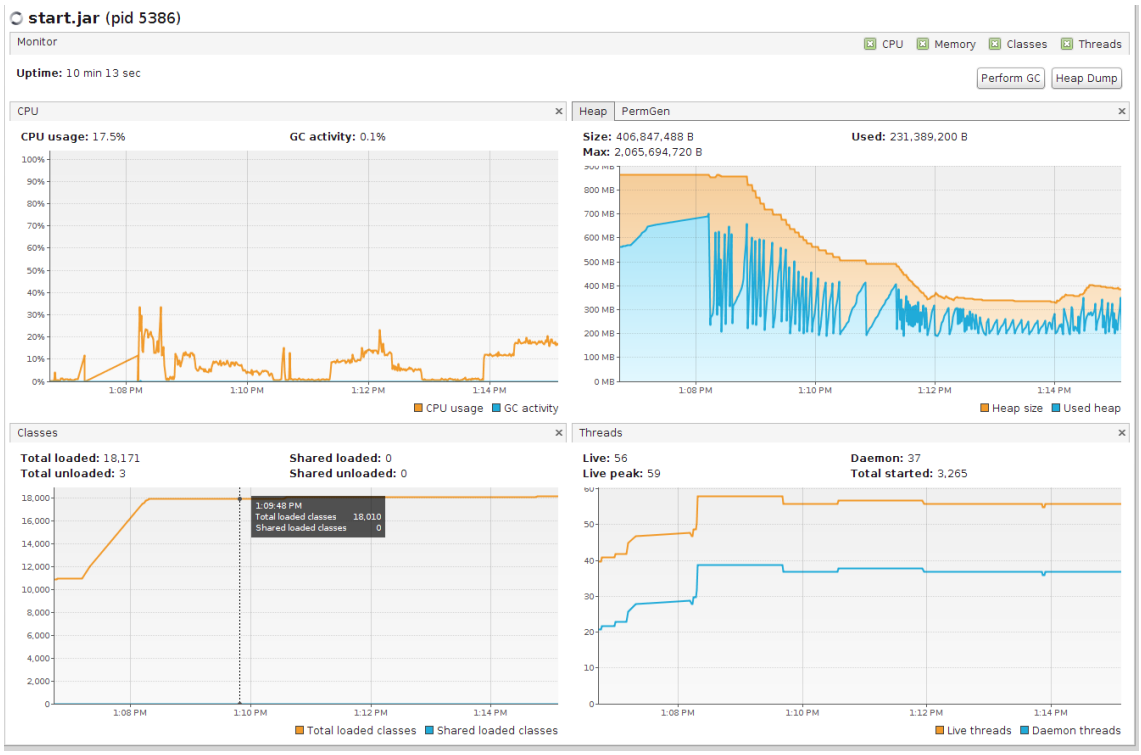

Figure A.12: Resources used when running Exp6 with 1000 messages per minute

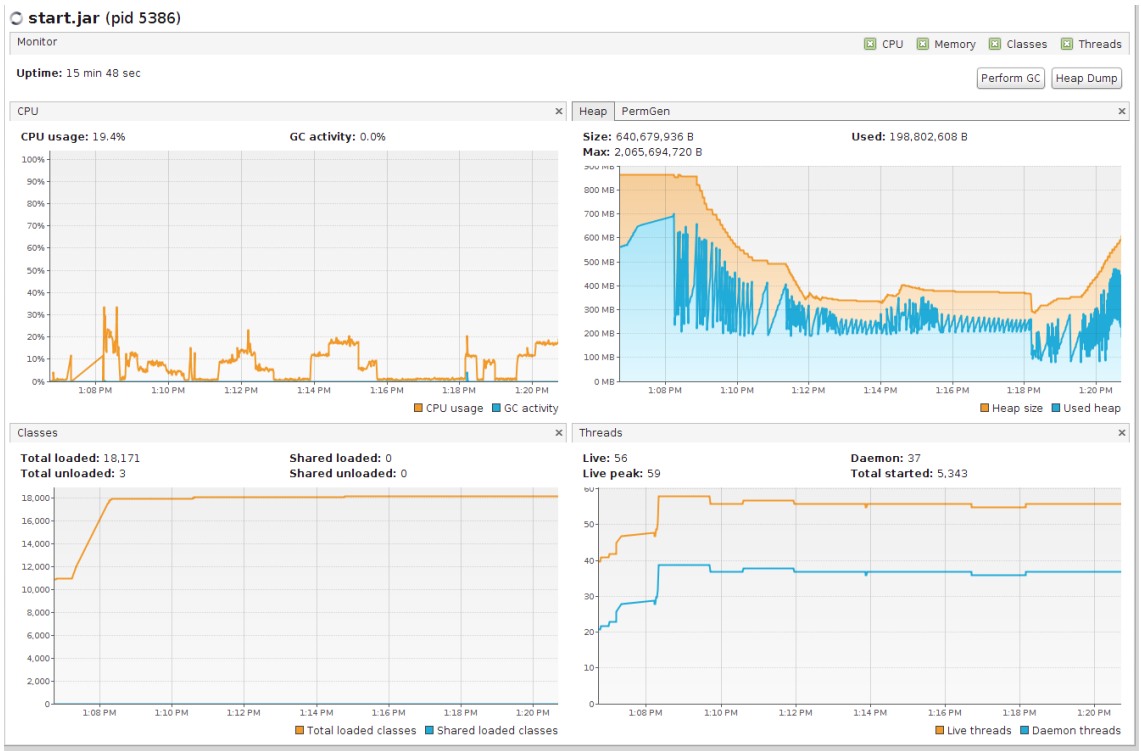

Figure A.13: Resources used when running Exp6 with 2000 messages per minute

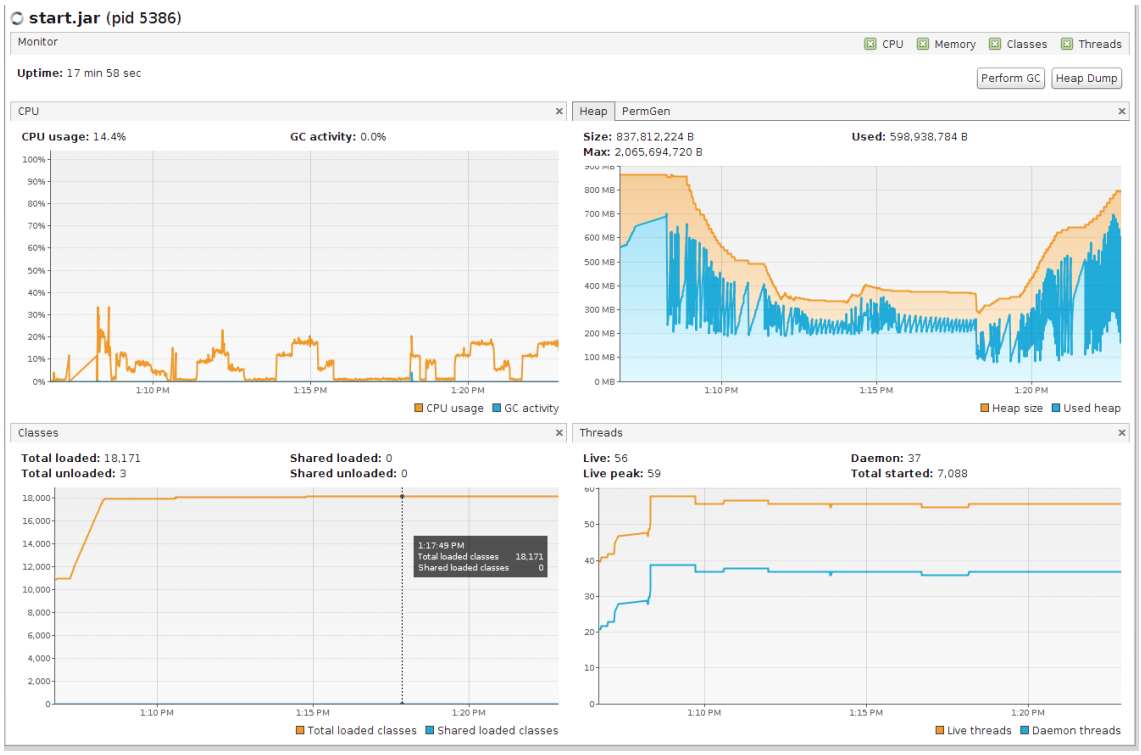

Figure A.14: Resources used when running Exp6 with 4000 messages per minute

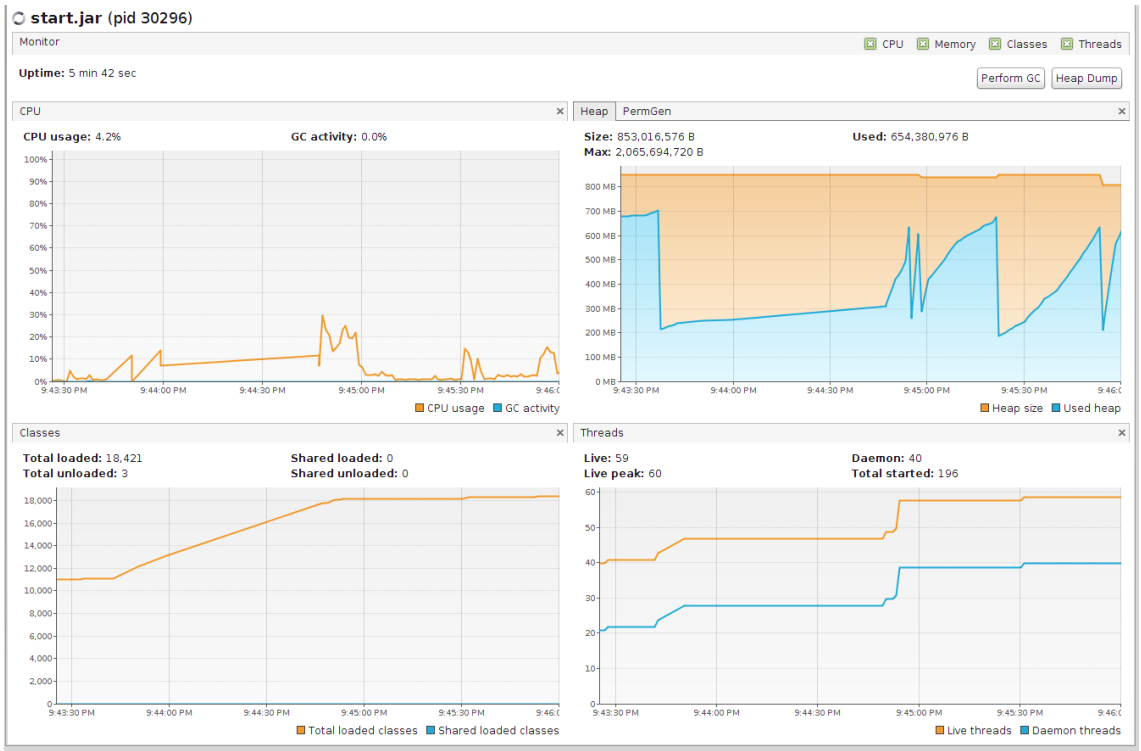

Figure A.15: Resources used when running Exp7 with 60 messages per minute

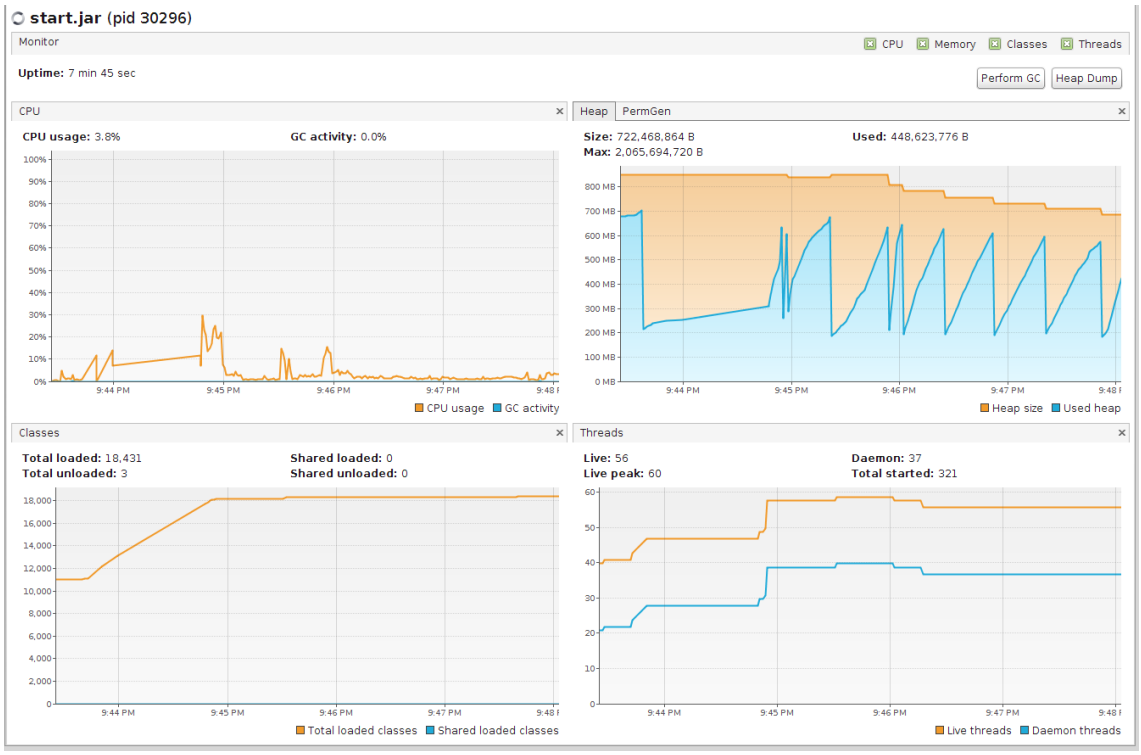

Figure A.16: Resources used when running Exp7 with 120 messages per minute

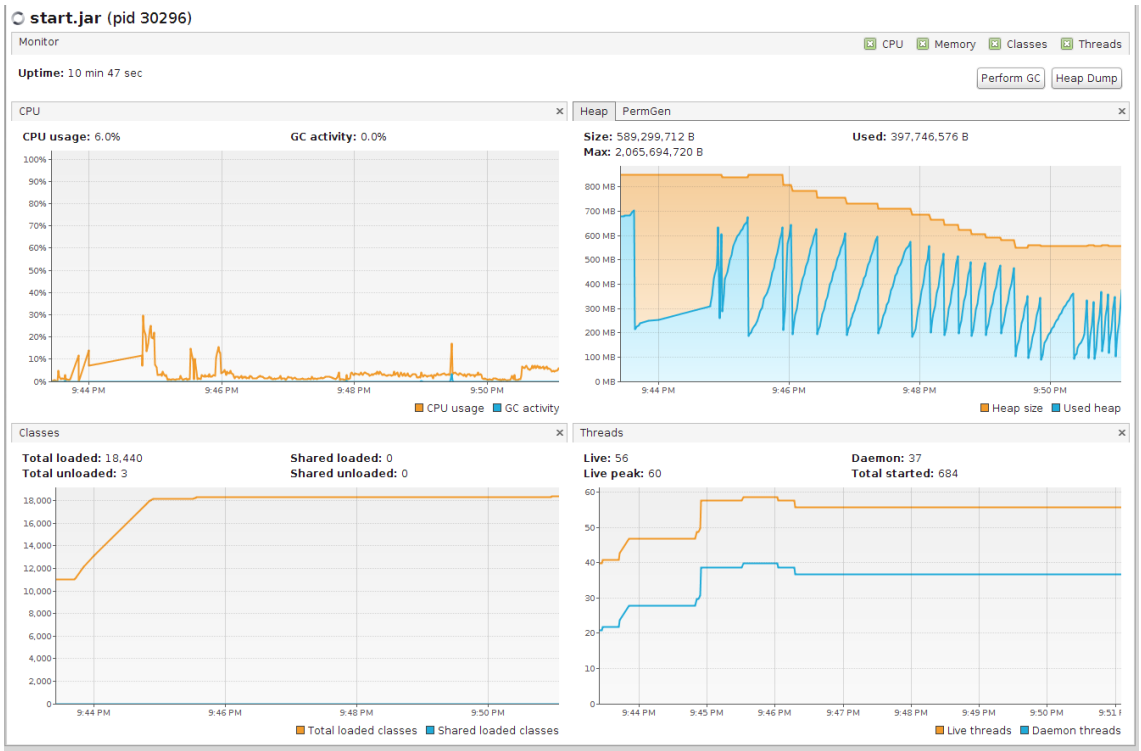

Figure A.17: Resources used when running Exp7 with 240 messages per minute

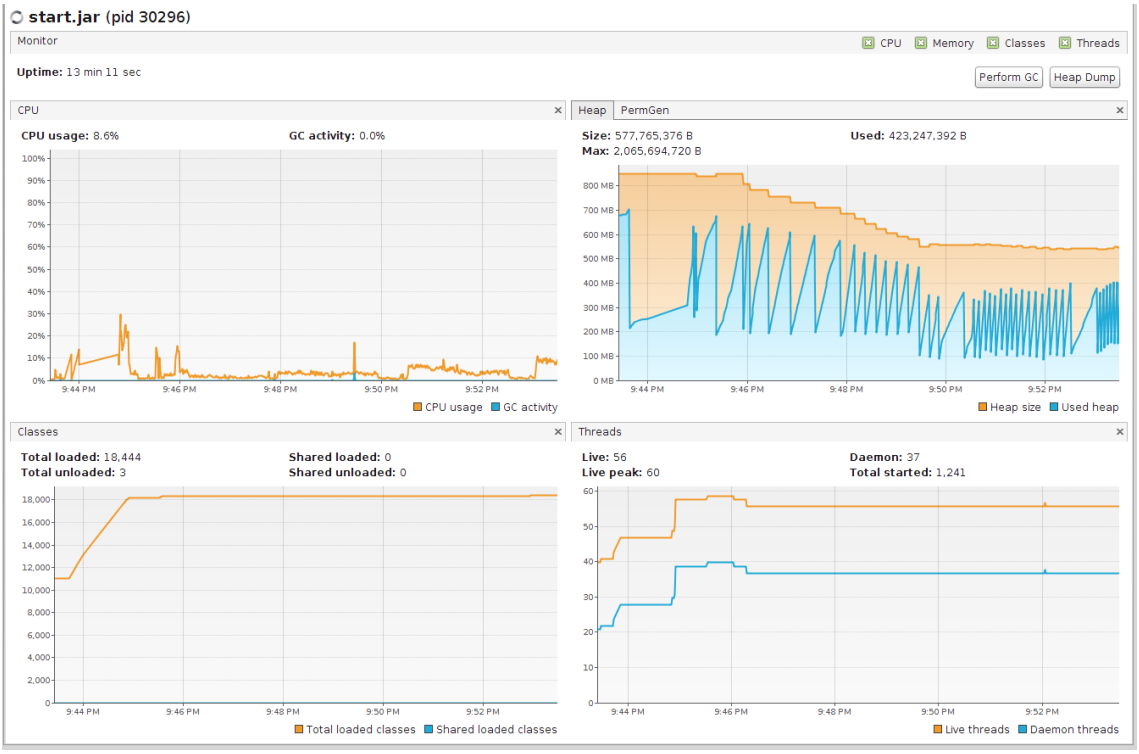

Figure A.18: Resources used when running Exp7 with 480 messages per minute

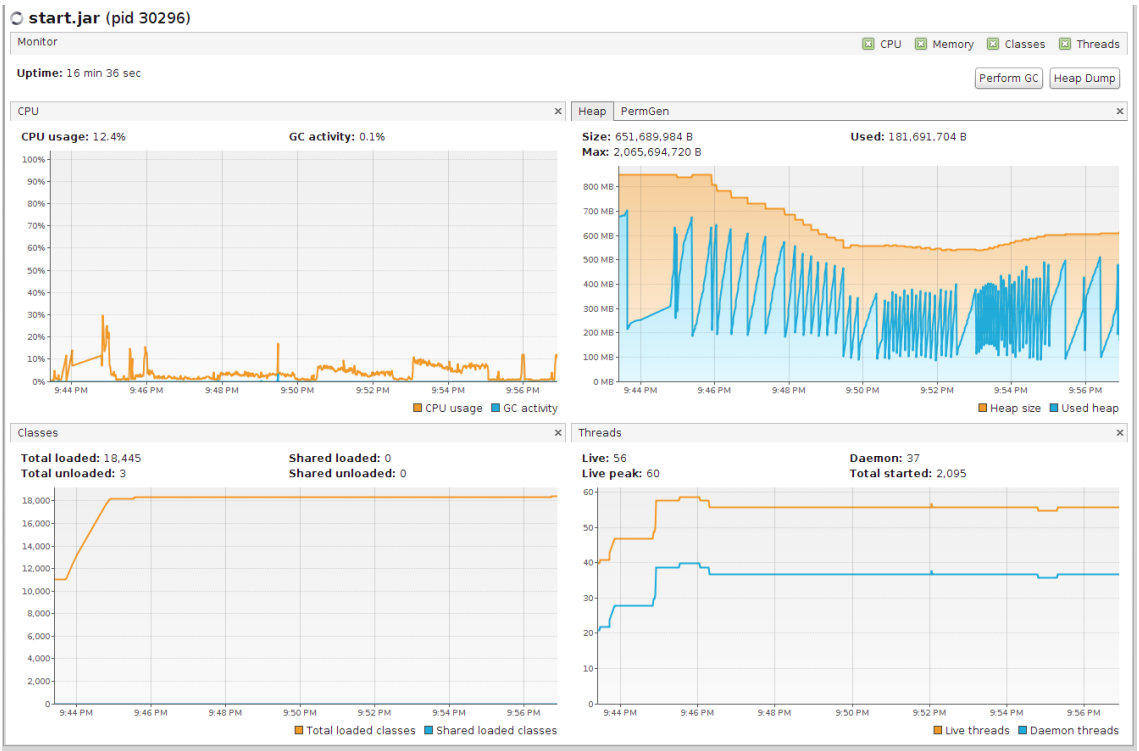

Figure A.19: Resources used when running Exp7 with 1000 messages per minute

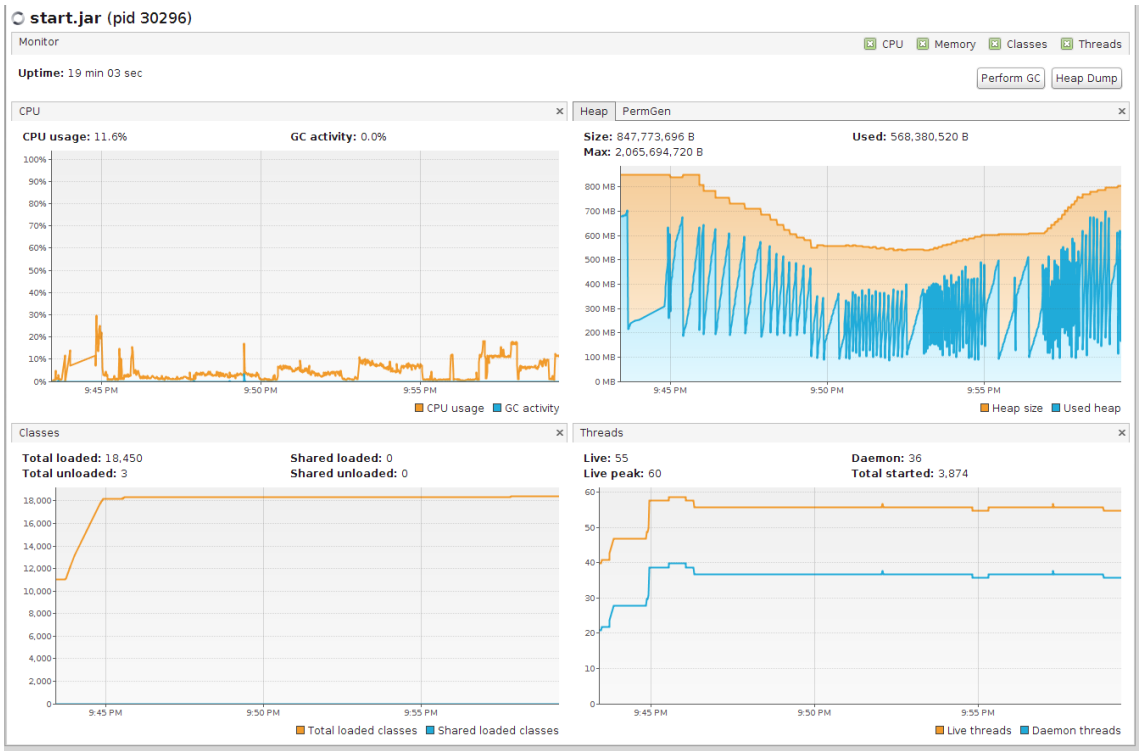

Figure A.20: Resources used when running Exp7 with 2000 messages per minute

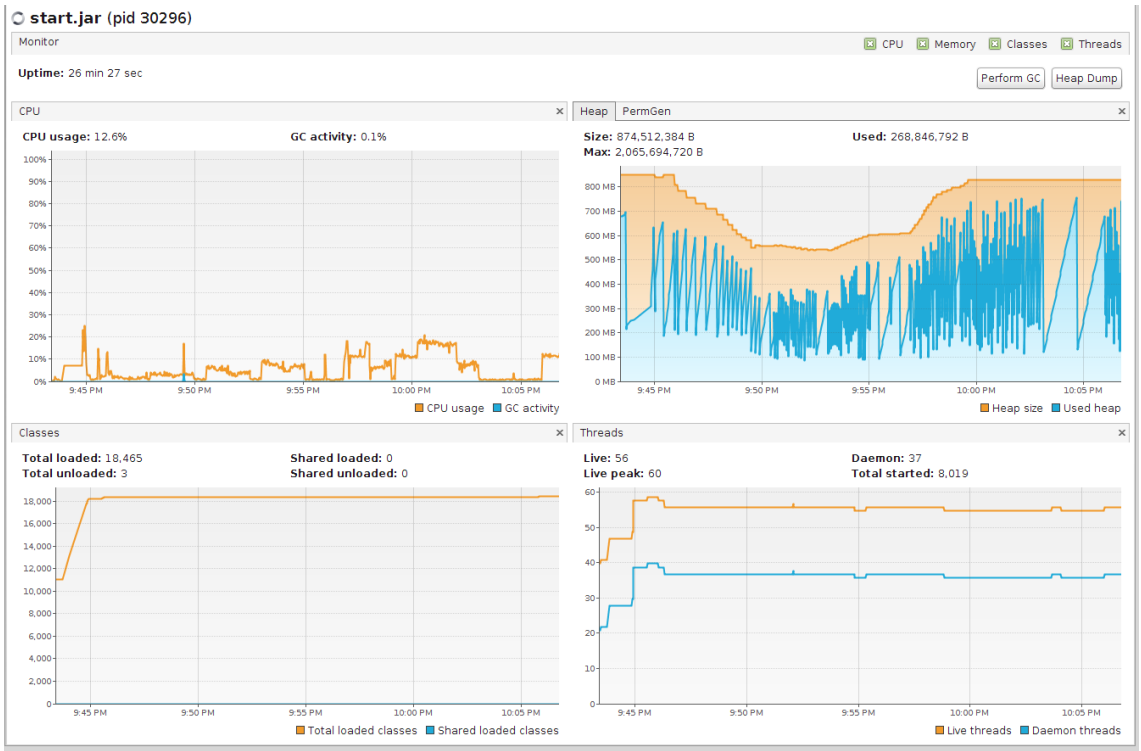

Figure A.21: Resources used when running Exp7 with 4000 messages per minute

A.1.2 visualVM CPU profiler screenshots

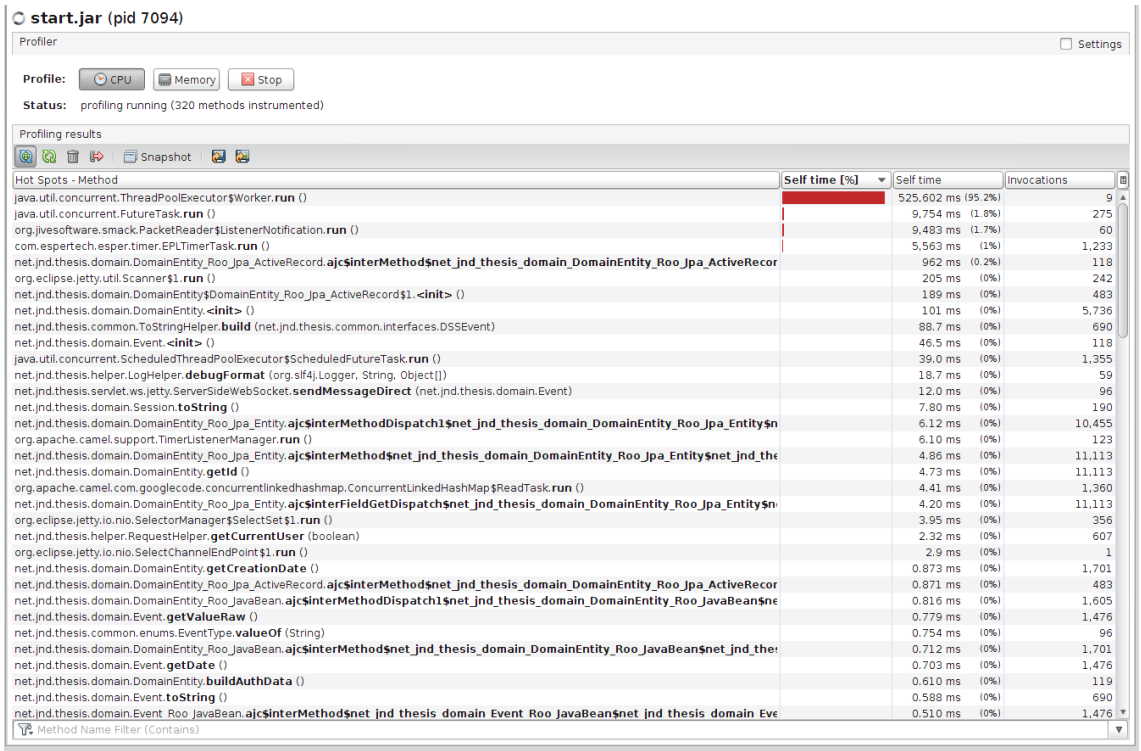

Figure A.22: CPU profiler for the case of 60 messages per minute using Exp6

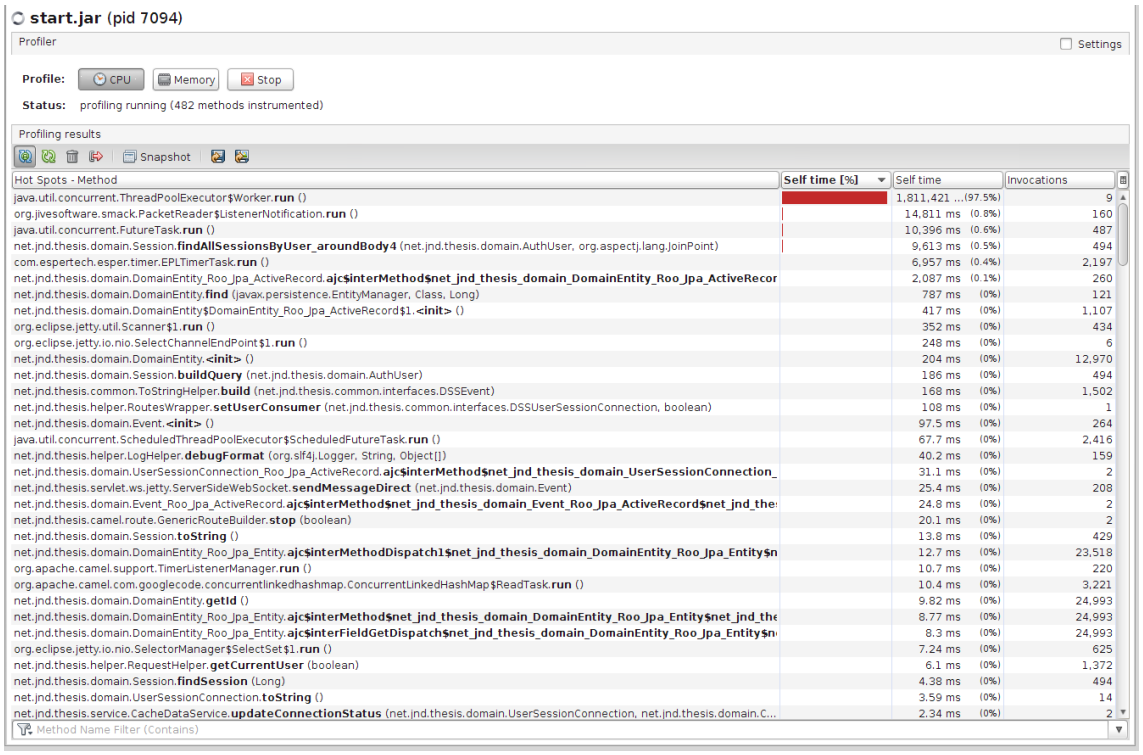

Figure A.23: CPU profiler for the case of 120 messages per minute using Exp6

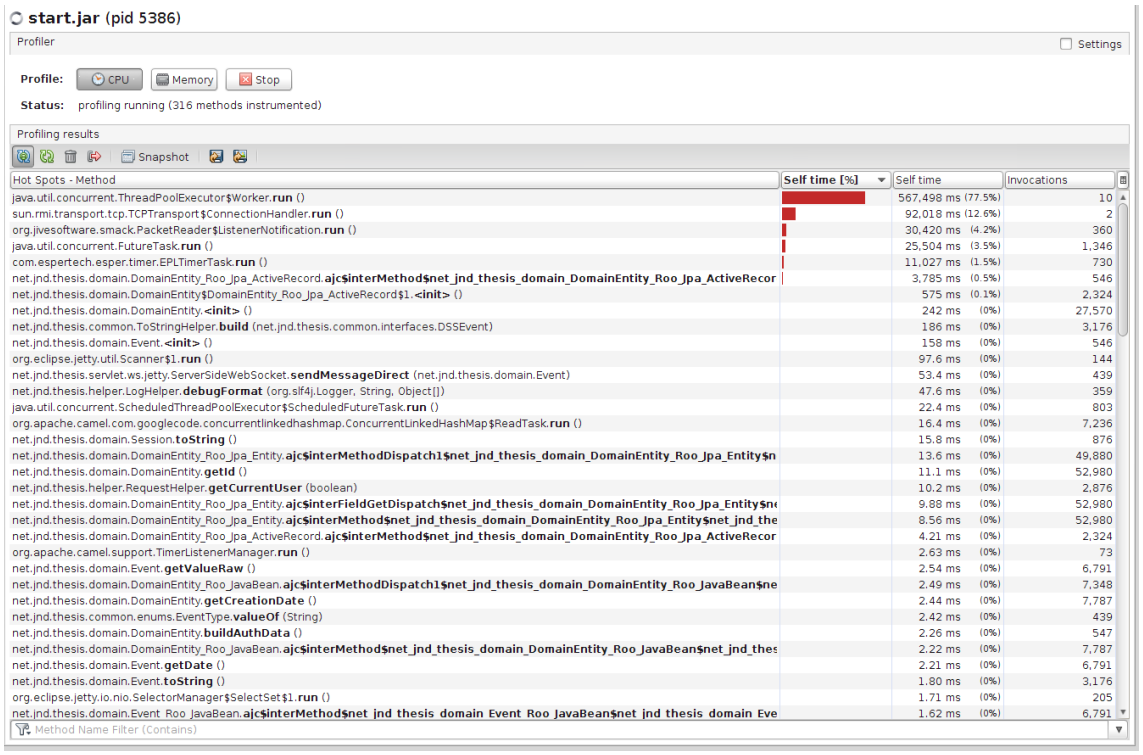

Figure A.24: CPU profiler for the case of 240 messages per minute using Exp6

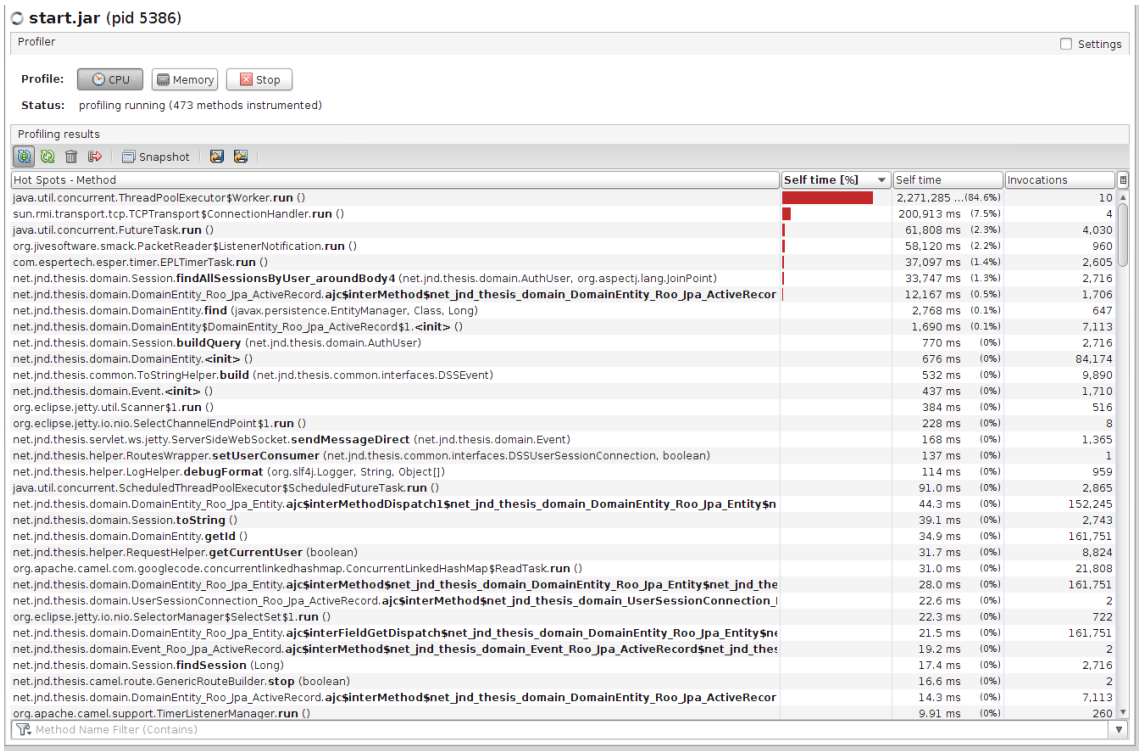

Figure A.25: CPU profiler for the case of 480 messages per minute using Exp6

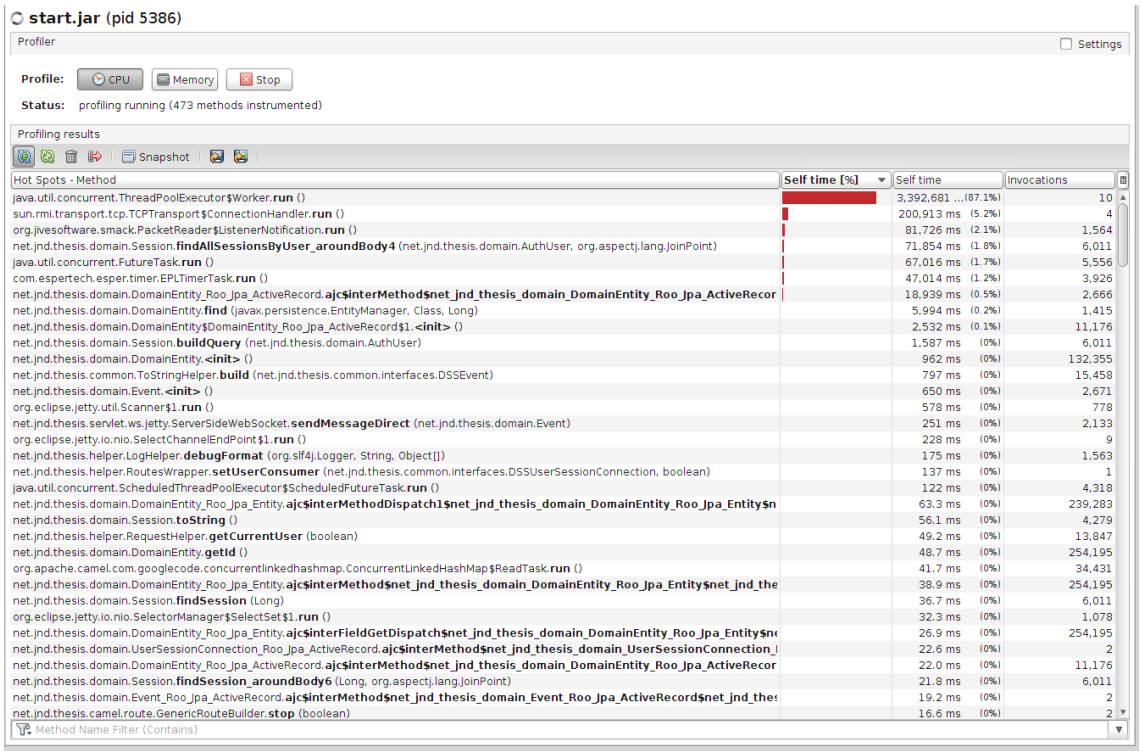

Figure A.26: CPU profiler for the case of 1000 messages per minute using Exp6

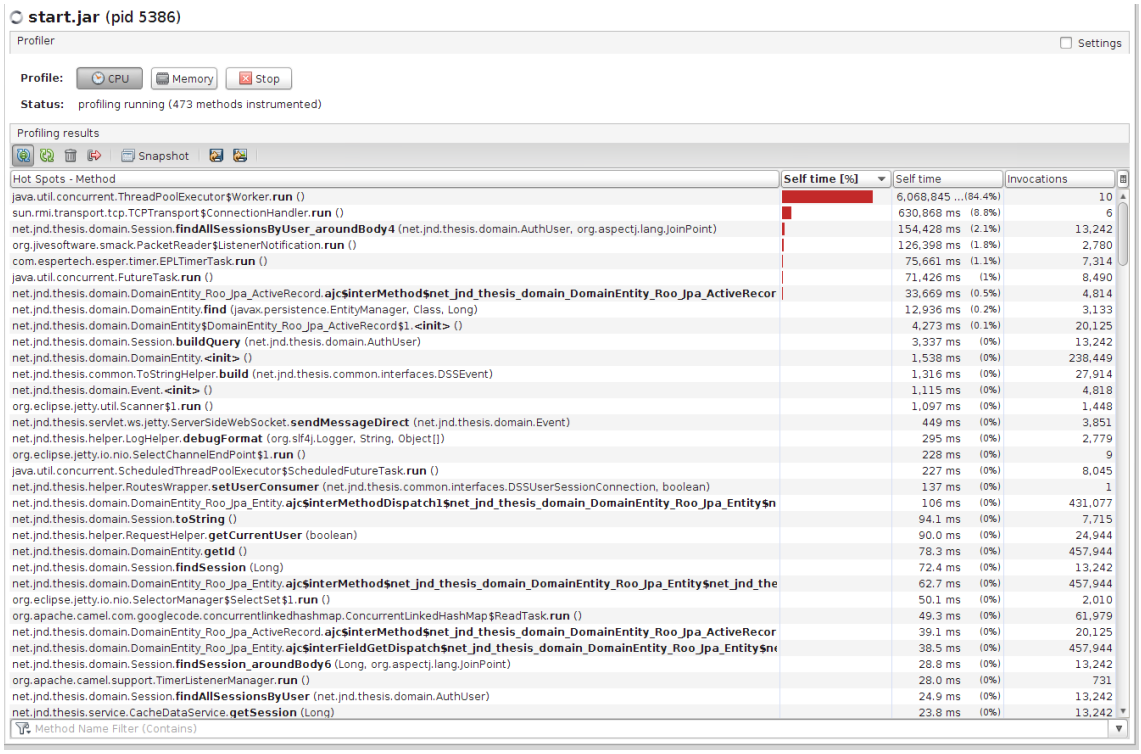

Figure A.27: CPU profiler for the case of 2000 messages per minute using Exp6

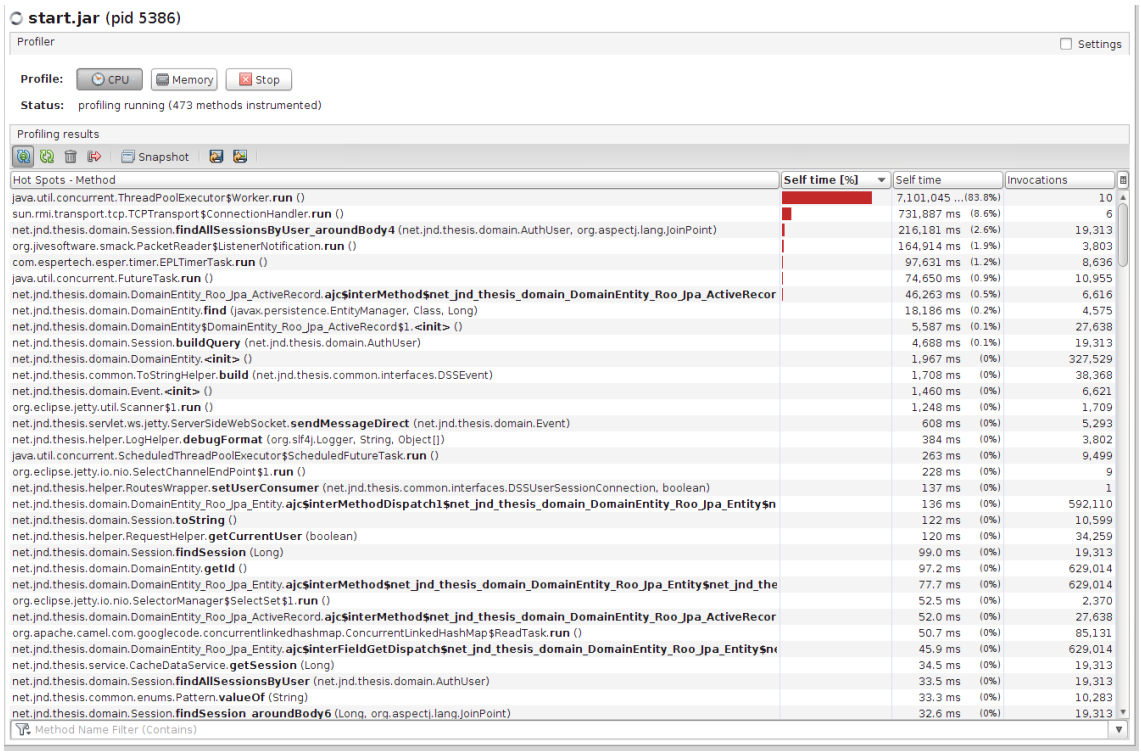

Figure A.28: CPU profiler for the case of 4000 messages per minute using Exp6

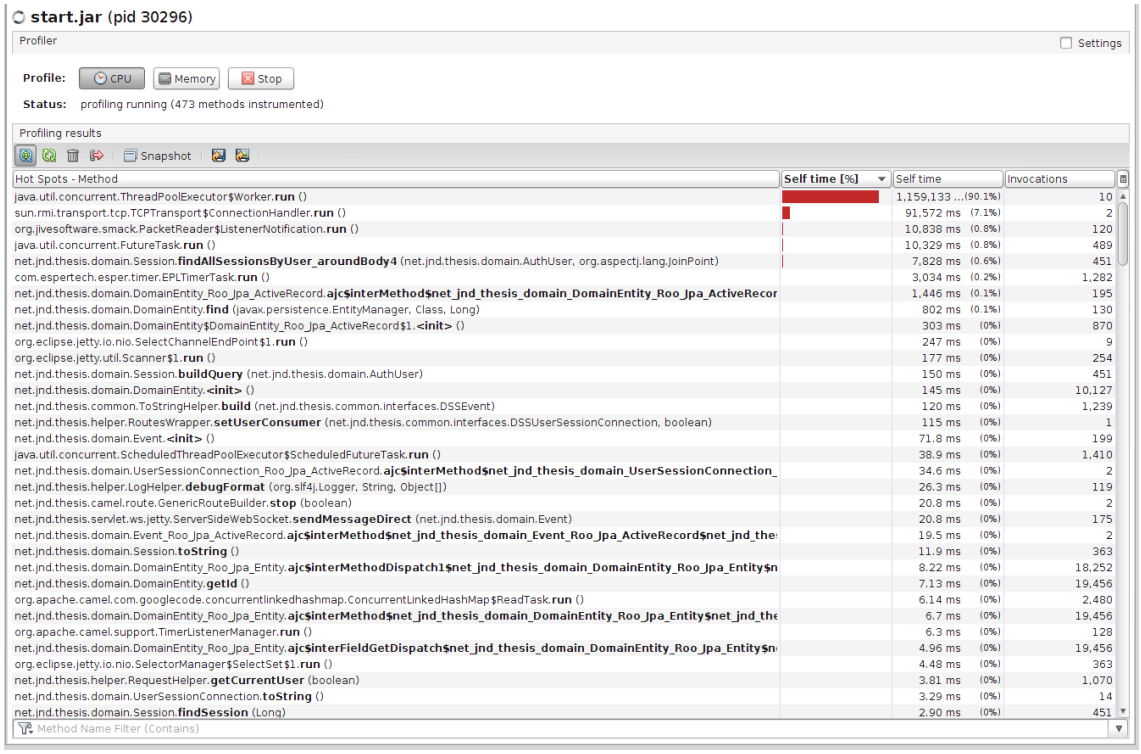

Figure A.29: CPU profiler for the case of 60 messages per minute using Exp7

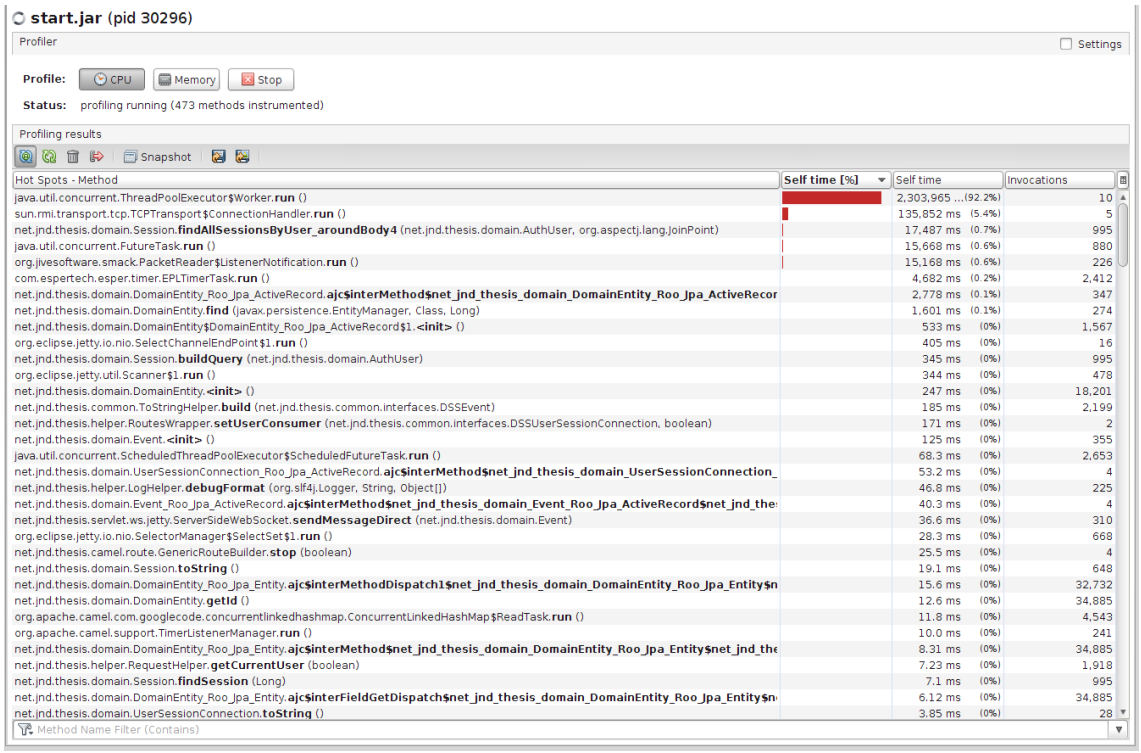

Figure A.30: CPU profiler for the case of 120 messages per minute using Exp7

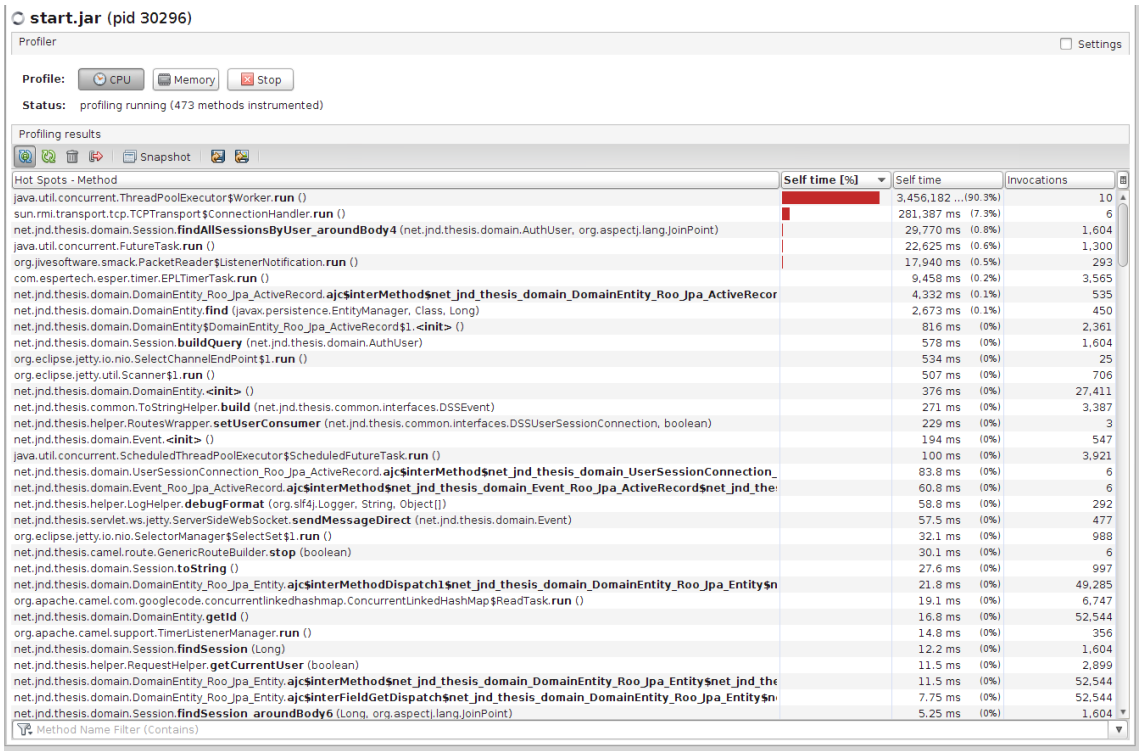

Figure A.31: CPU profiler for the case of 240 messages per minute using Exp7

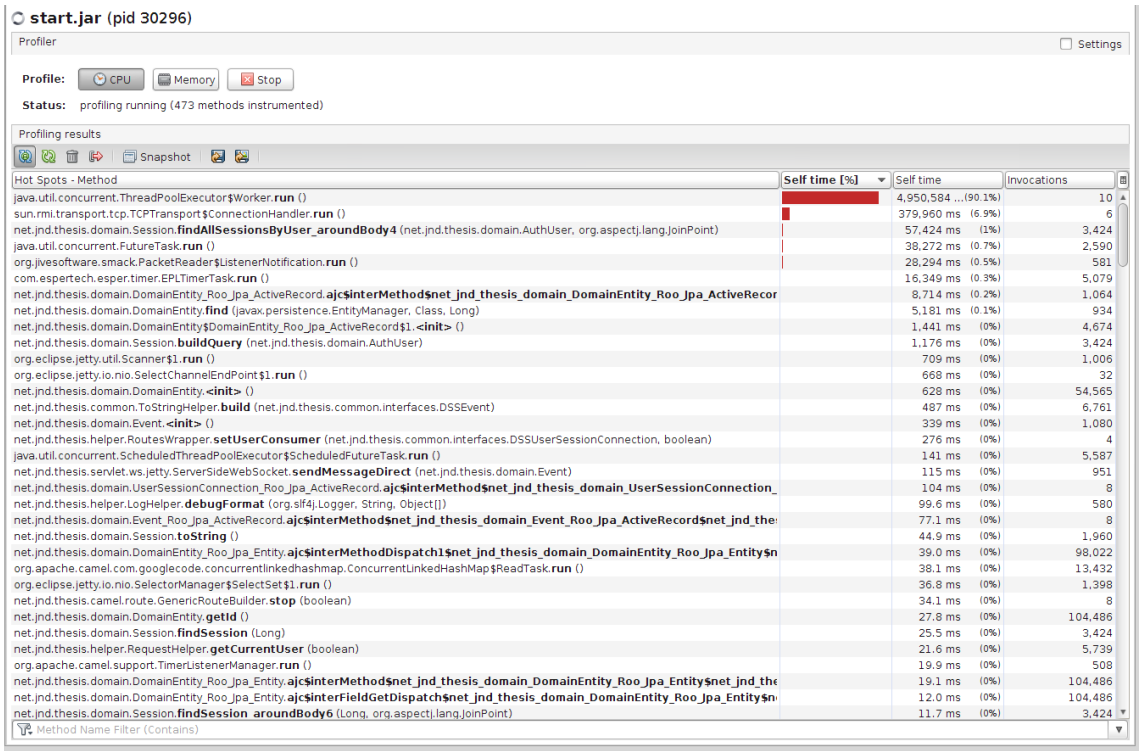

Figure A.32: CPU profiler for the case of 480 messages per minute using Exp7

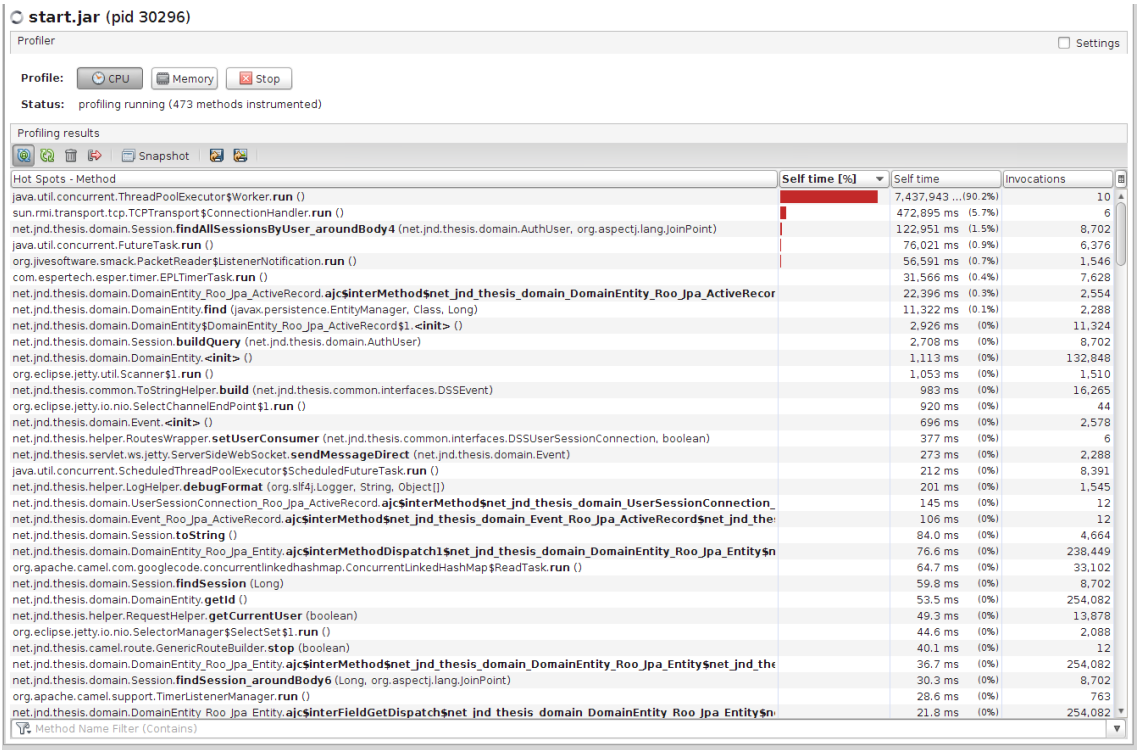

Figure A.33: CPU profiler for the case of 1000 messages per minute using Exp7

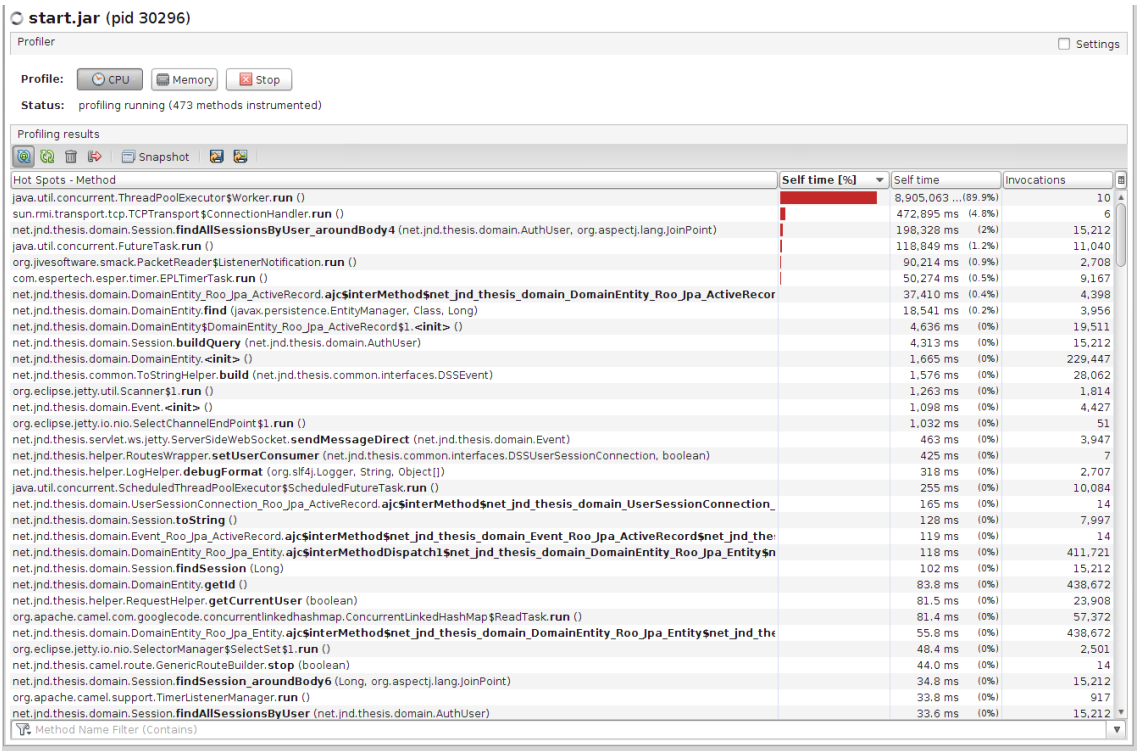

Figure A.34: CPU profiler for the case of 2000 messages per minute using Exp7

A.1.3 visualVM CPU sampler screenshots

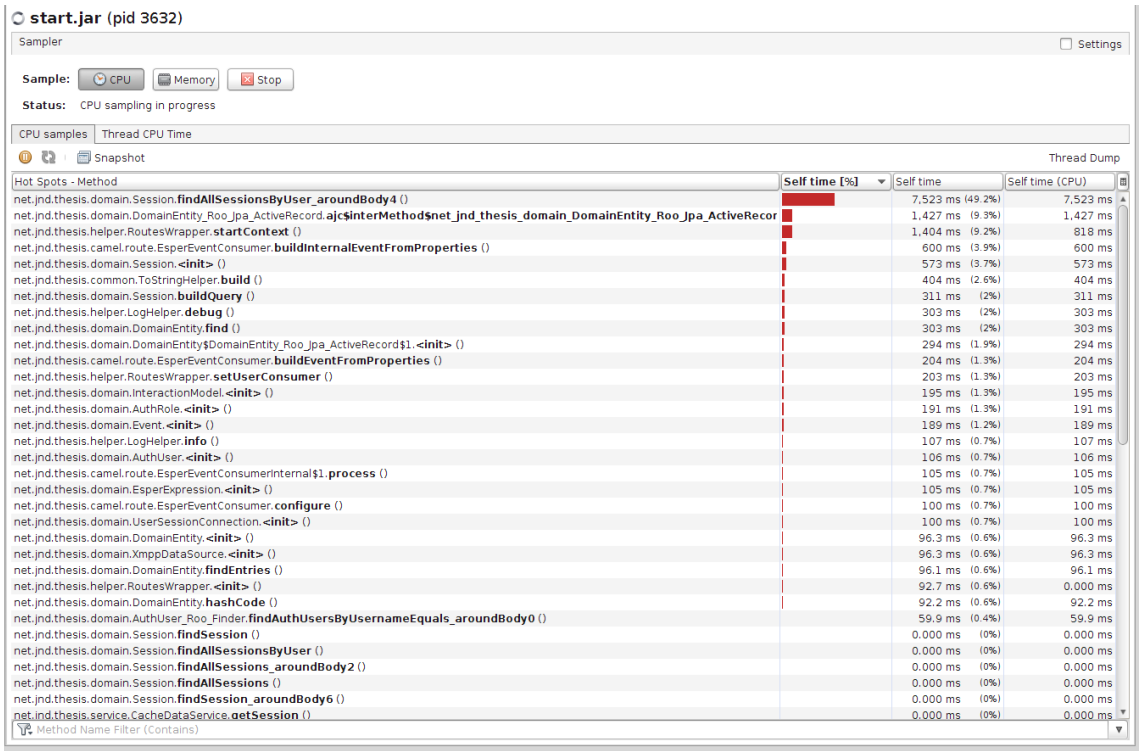

Figure A.35: CPU sampler for the case of 60 messages per minute using Exp6

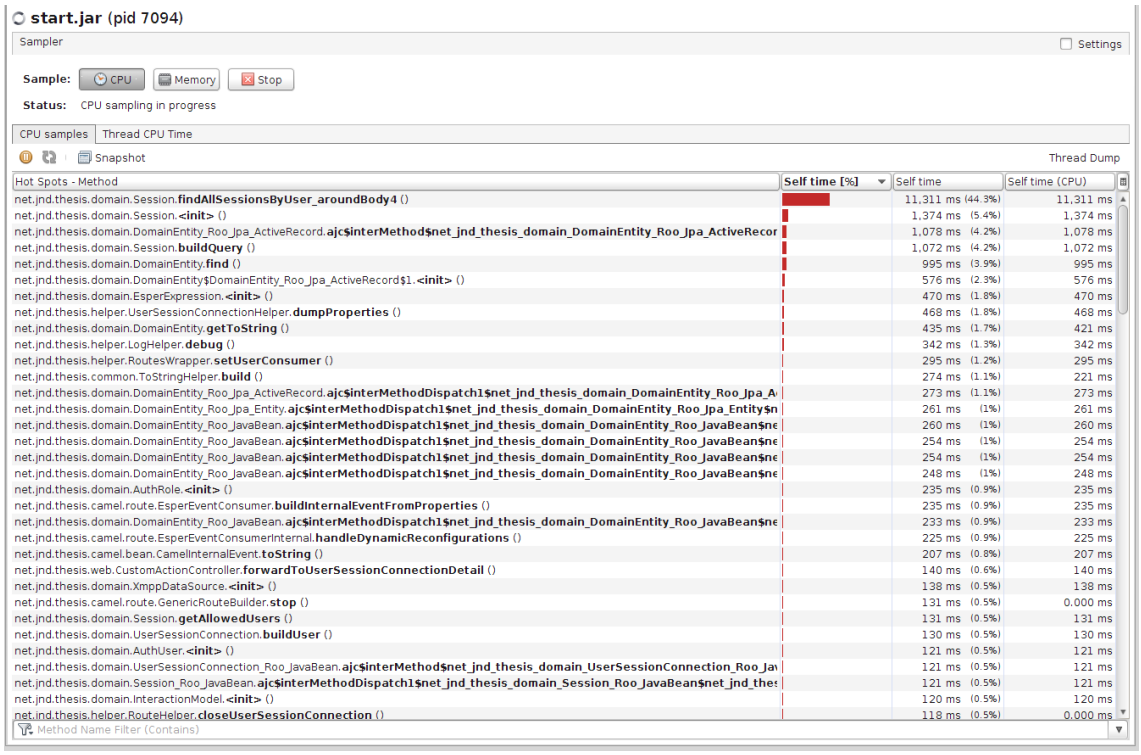

Figure A.36: CPU sampler for the case of 120 messages per minute using Exp6

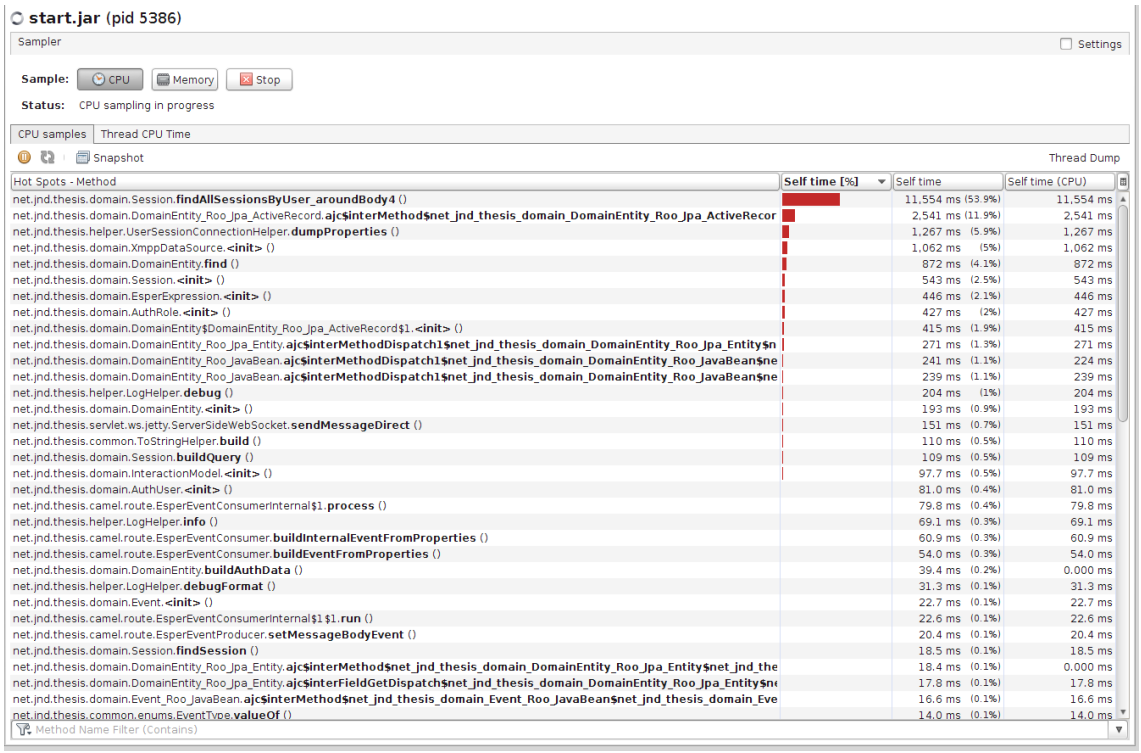

Figure A.37: CPU sampler for the case of 240 messages per minute using Exp6

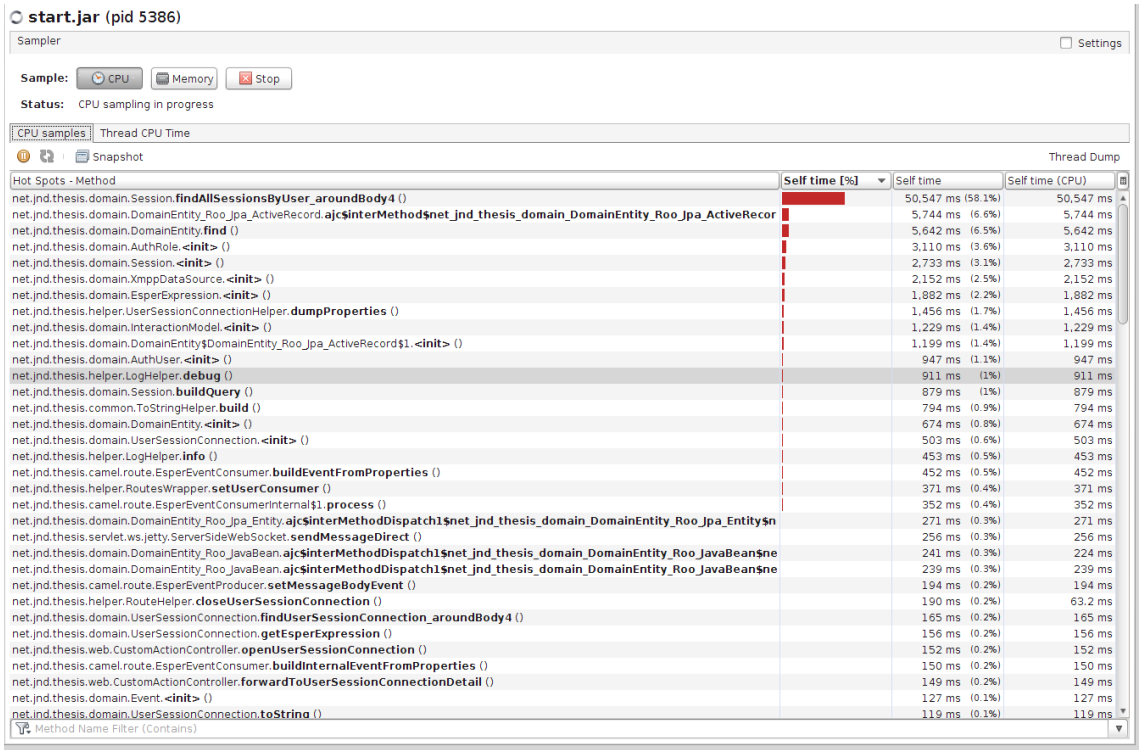

Figure A.38: CPU sampler for the case of 480 messages per minute using Exp6

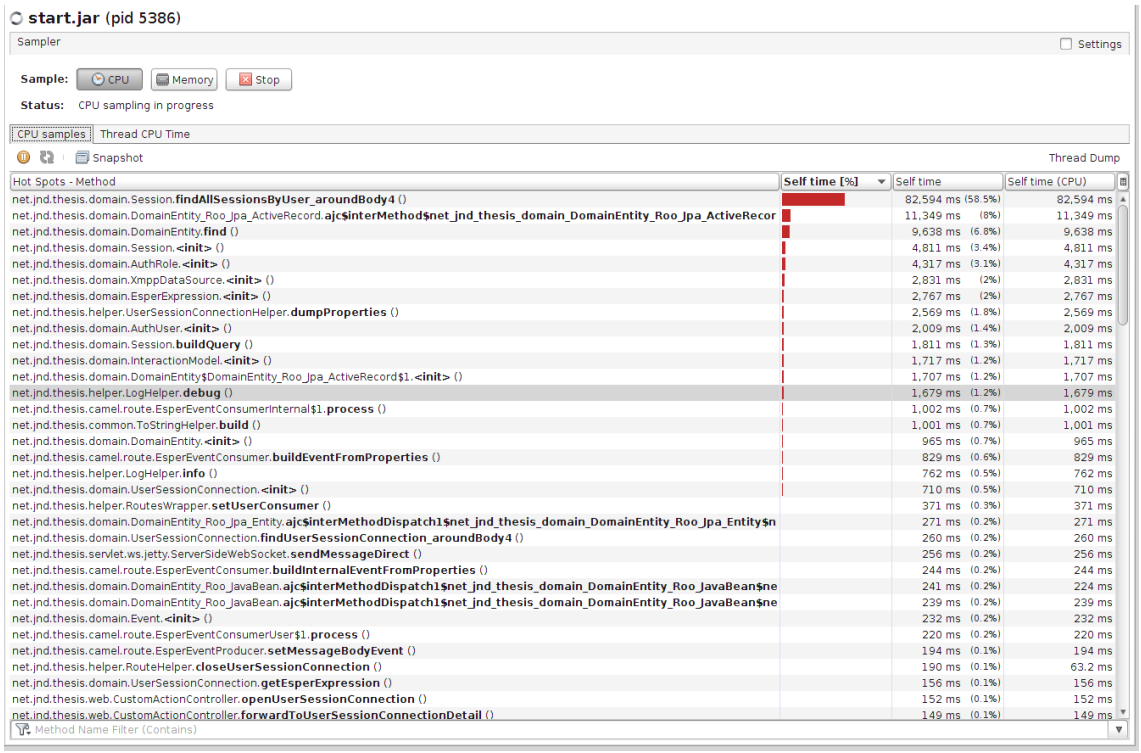

Figure A.39: CPU sampler for the case of 1000 messages per minute using Exp6

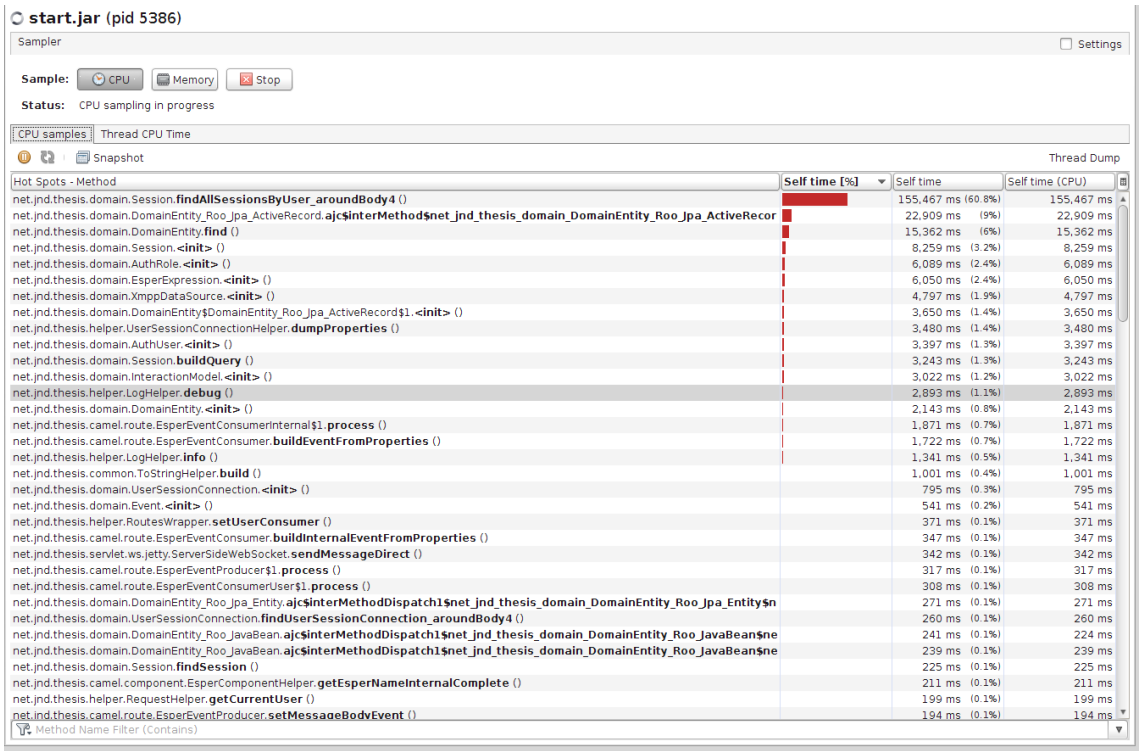

Figure A.40: CPU sampler for the case of 2000 messages per minute using Exp6

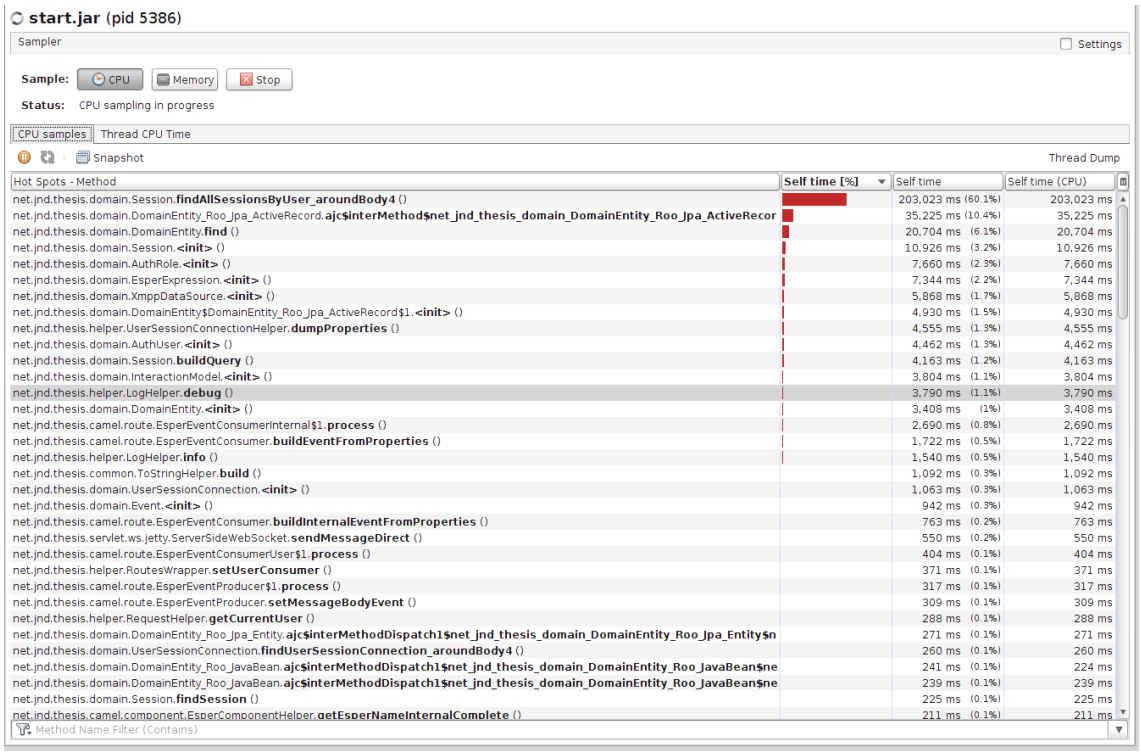

Figure A.41: CPU sampler for the case of 4000 messages per minute using Exp6

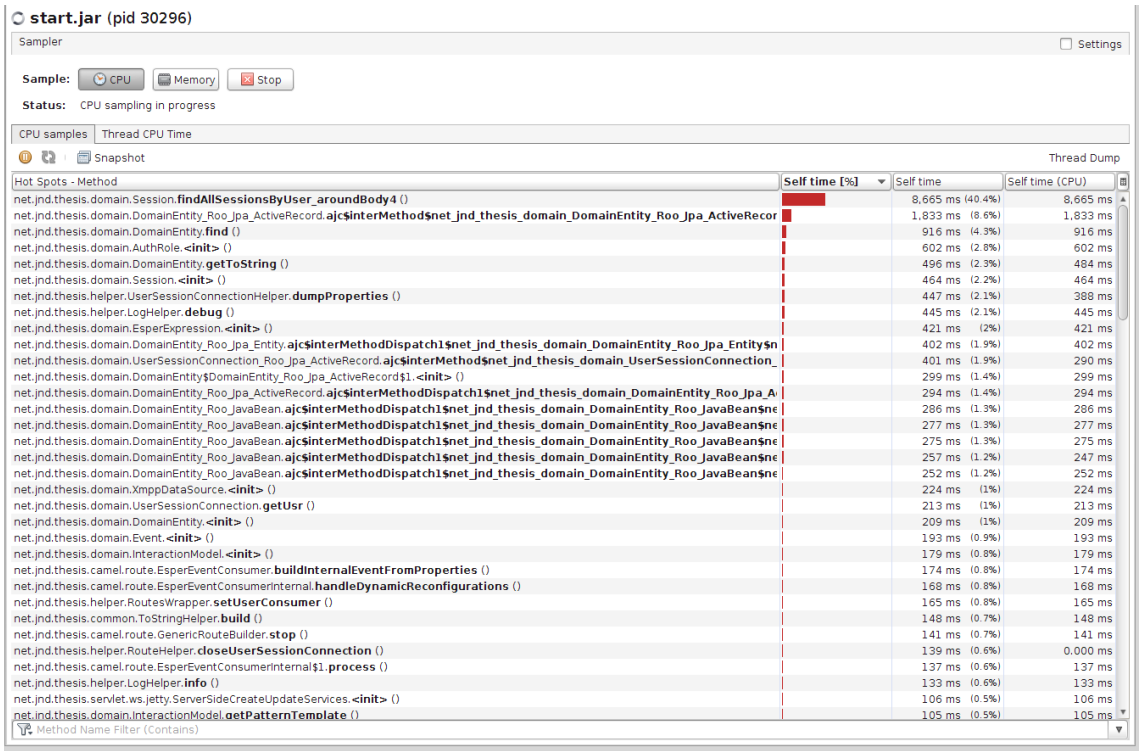

Figure A.42: CPU sampler for the case of 60 messages per minute using Exp7

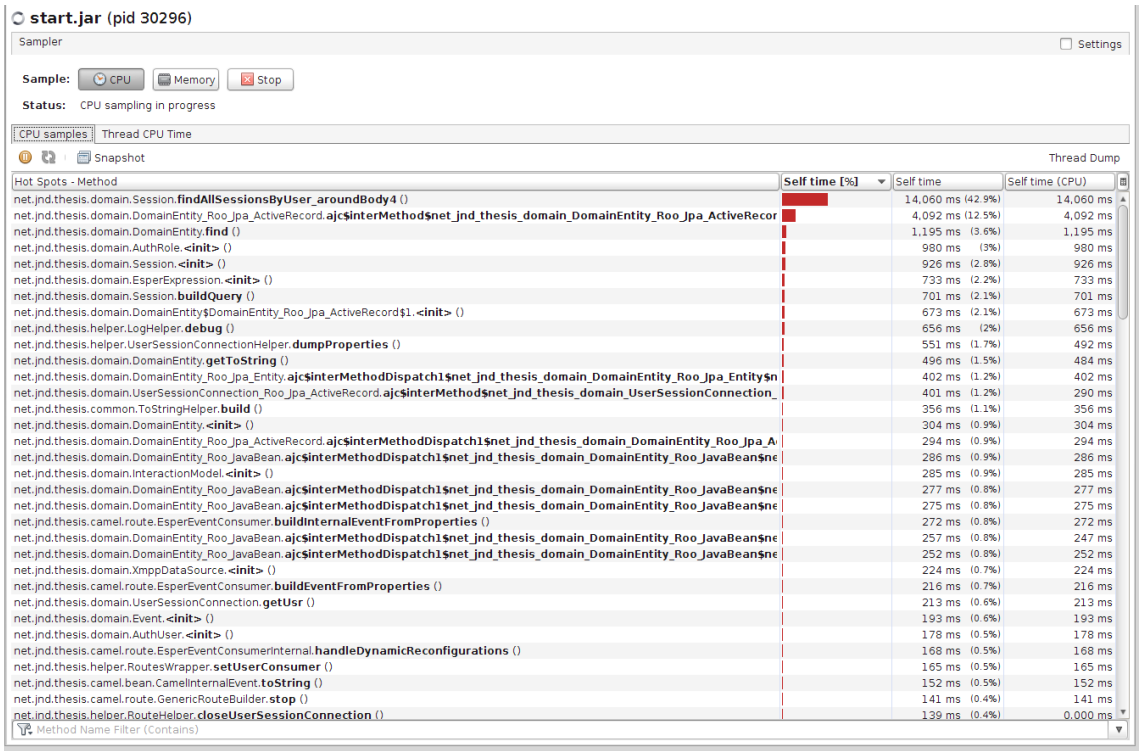

Figure A.43: CPU sampler for the case of 120 messages per minute using Exp7

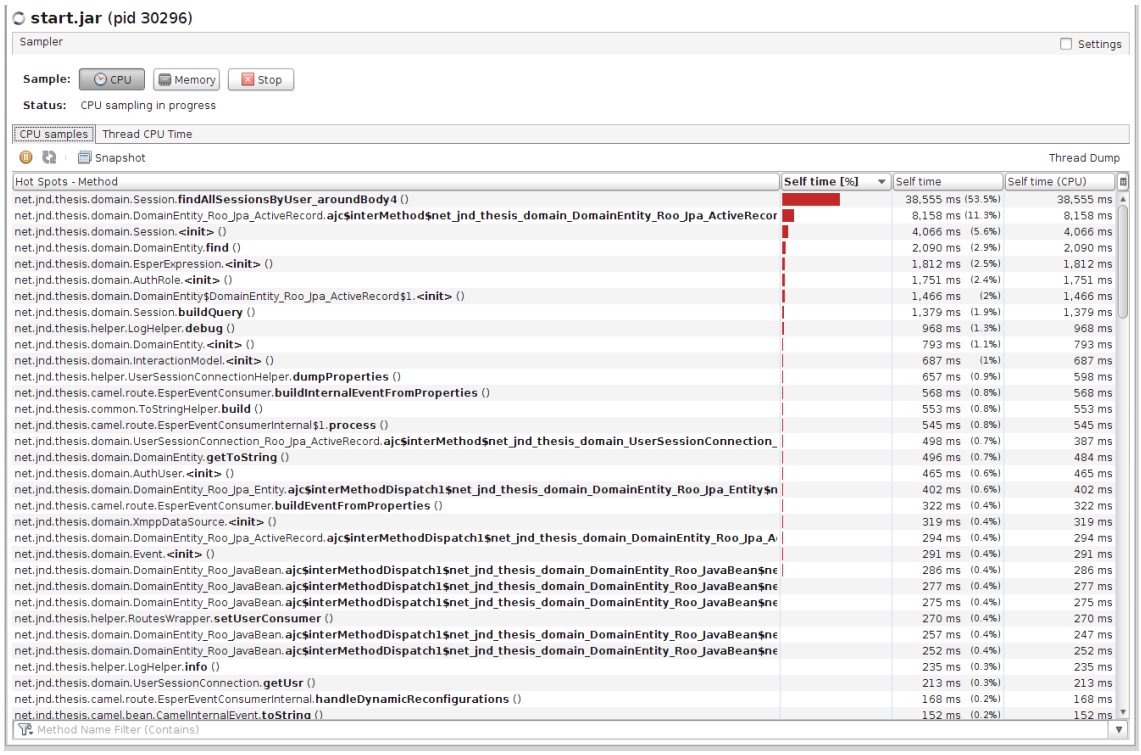

Figure A.44: CPU sampler for the case of 240 messages per minute using Exp7

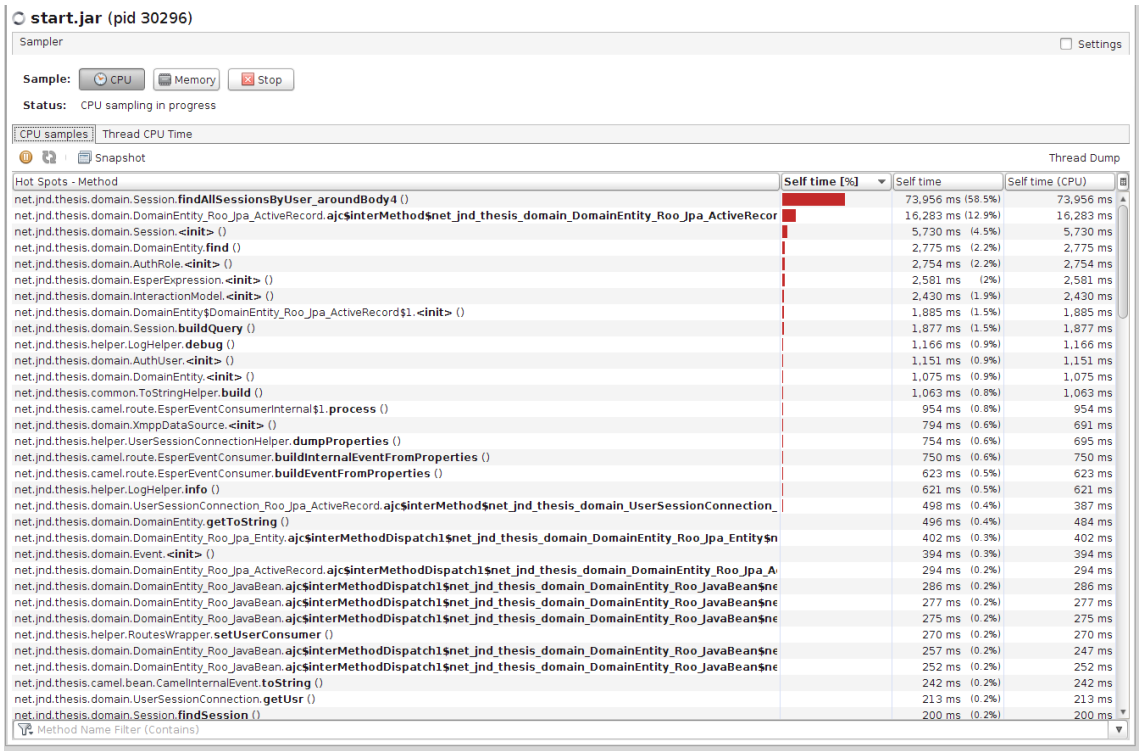

Figure A.45: CPU sampler for the case of 480 messages per minute using Exp7

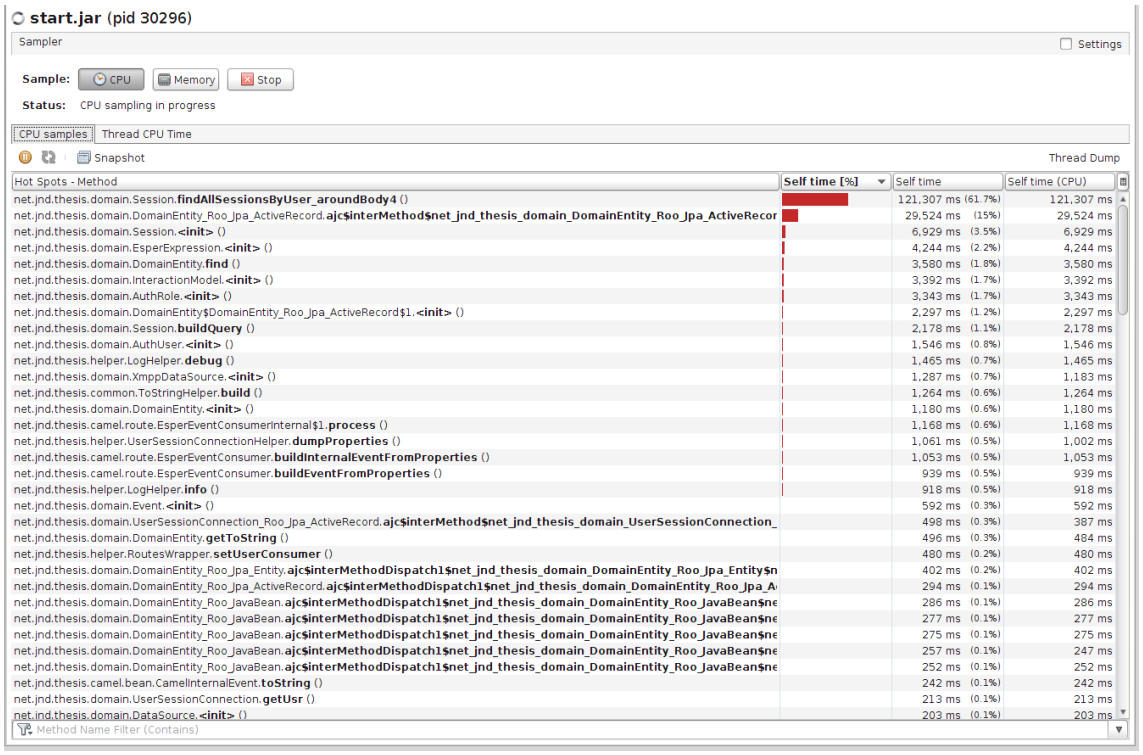

Figure A.46: CPU sampler for the case of 1000 messages per minute using Exp7

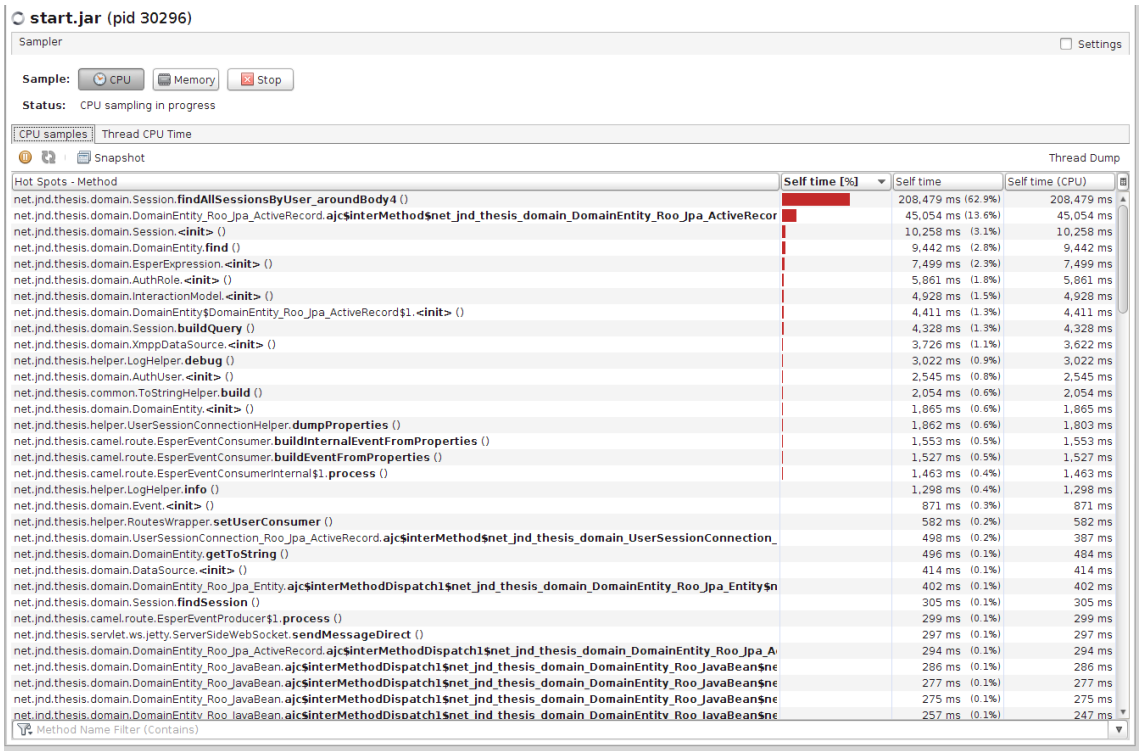

Figure A.47: CPU sampler for the case of 2000 messages per minute using Exp7

A.1.4 visualVM CPU threads screenshots

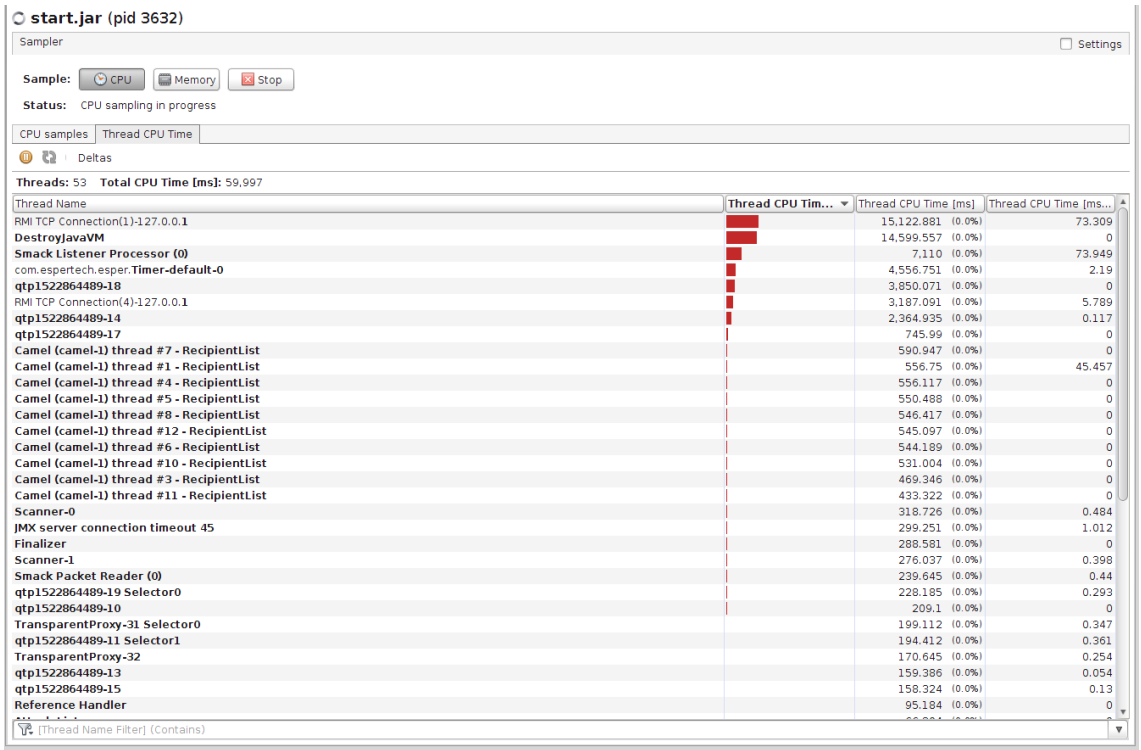

Figure A.48: CPU threads for the case of 60 messages per minute using Exp6

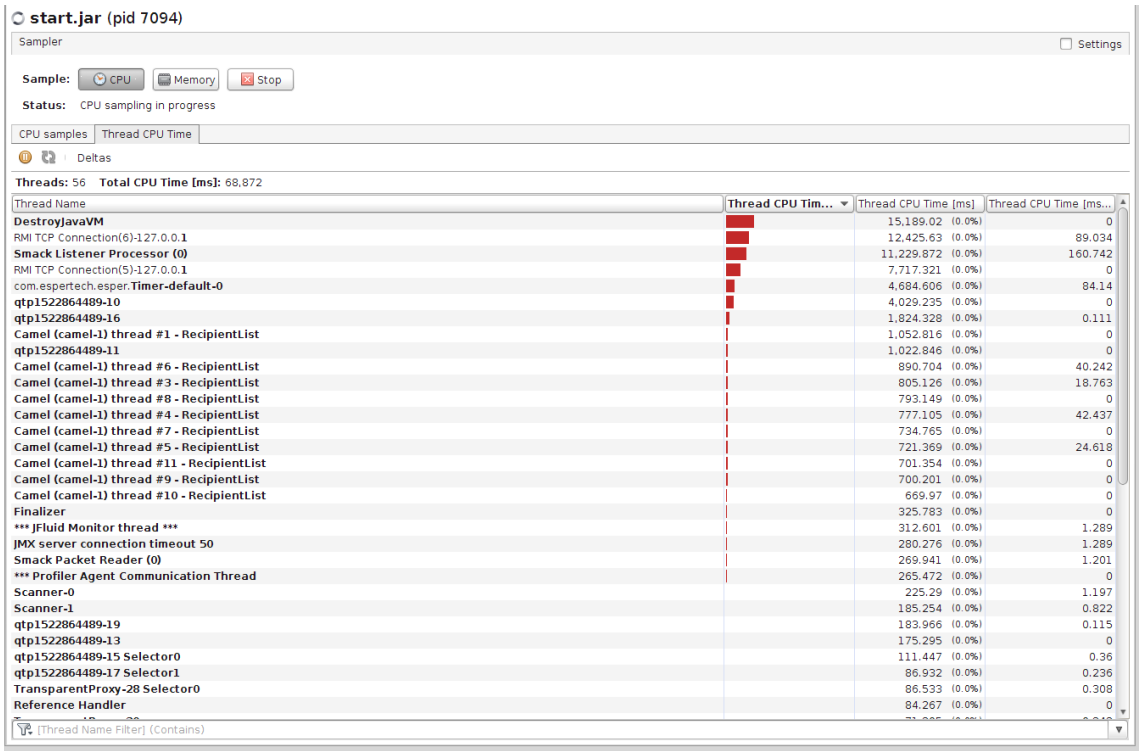

Figure A.49: CPU threads for the case of 120 messages per minute using Exp6

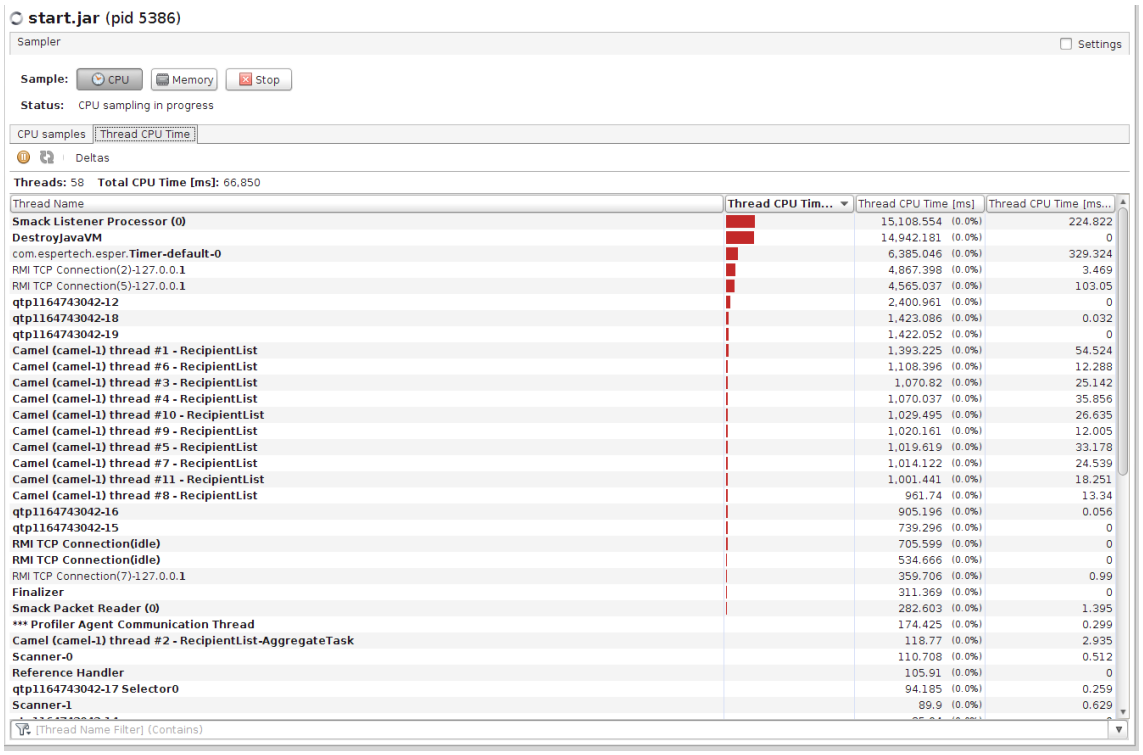

Figure A.50: CPU threads for the case of 240 messages per minute using Exp6

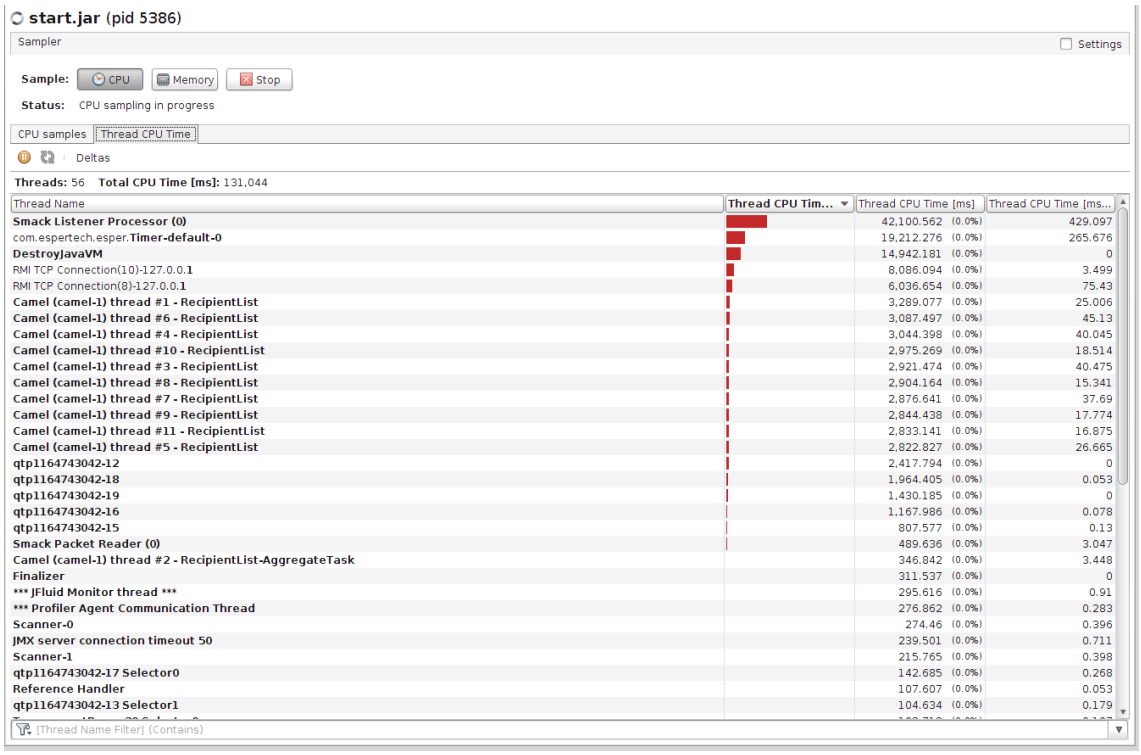

Figure A.51: CPU threads for the case of 480 messages per minute using Exp6

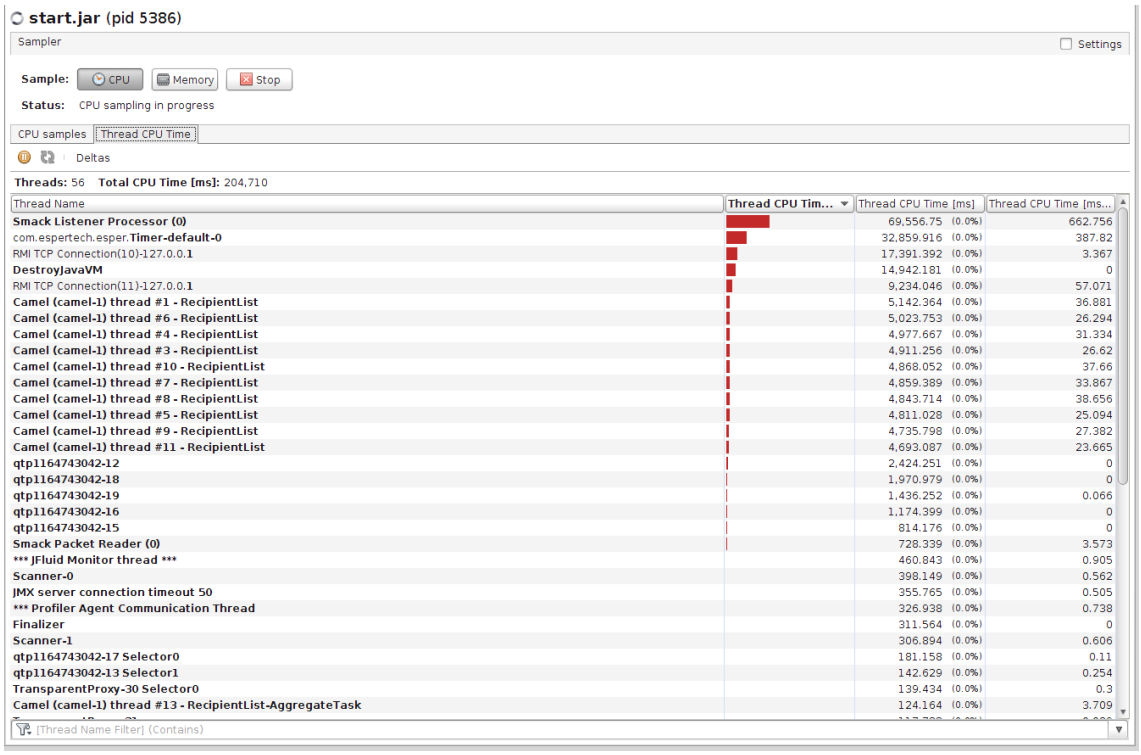

Figure A.52: CPU threads for the case of 1000 messages per minute using Exp6

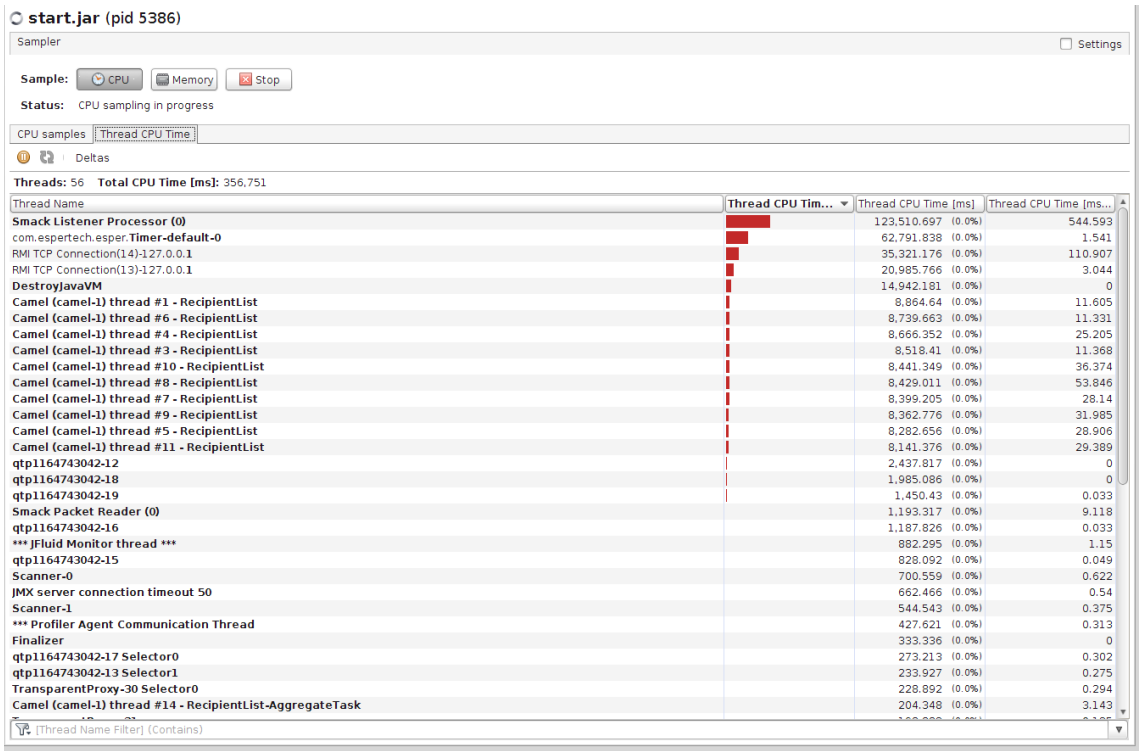

Figure A.53: CPU threads for the case of 2000 messages per minute using Exp6

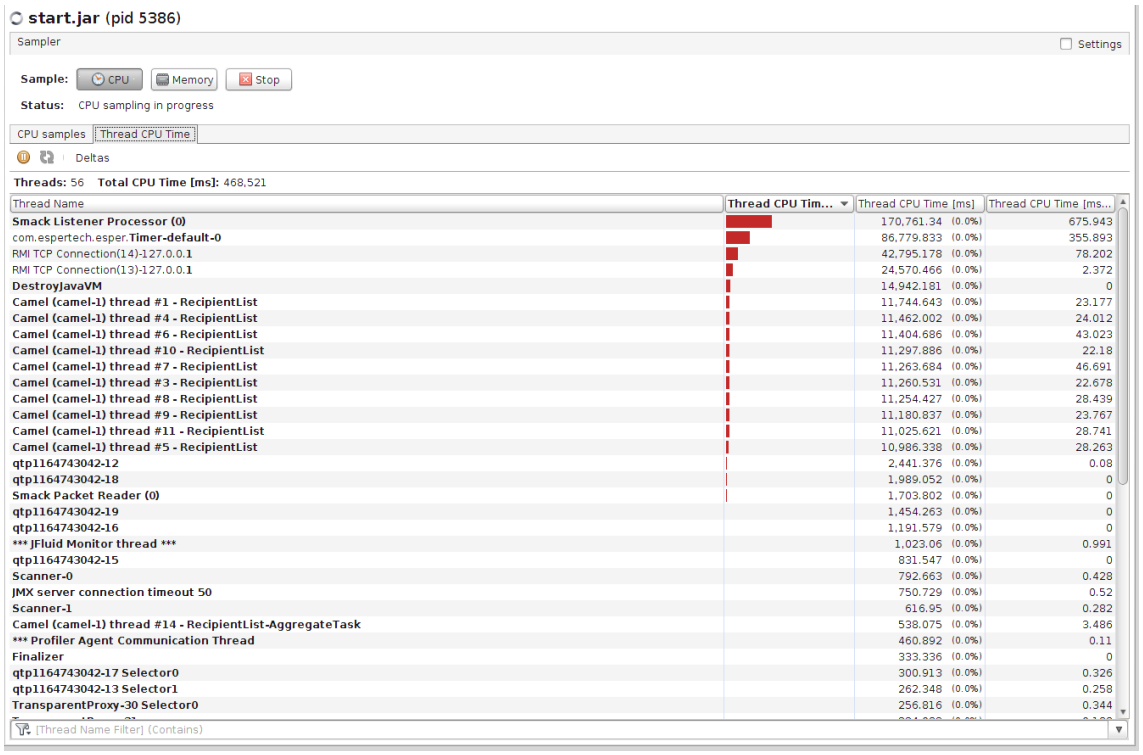

Figure A.54: CPU threads for the case of 4000 messages per minute using Exp6

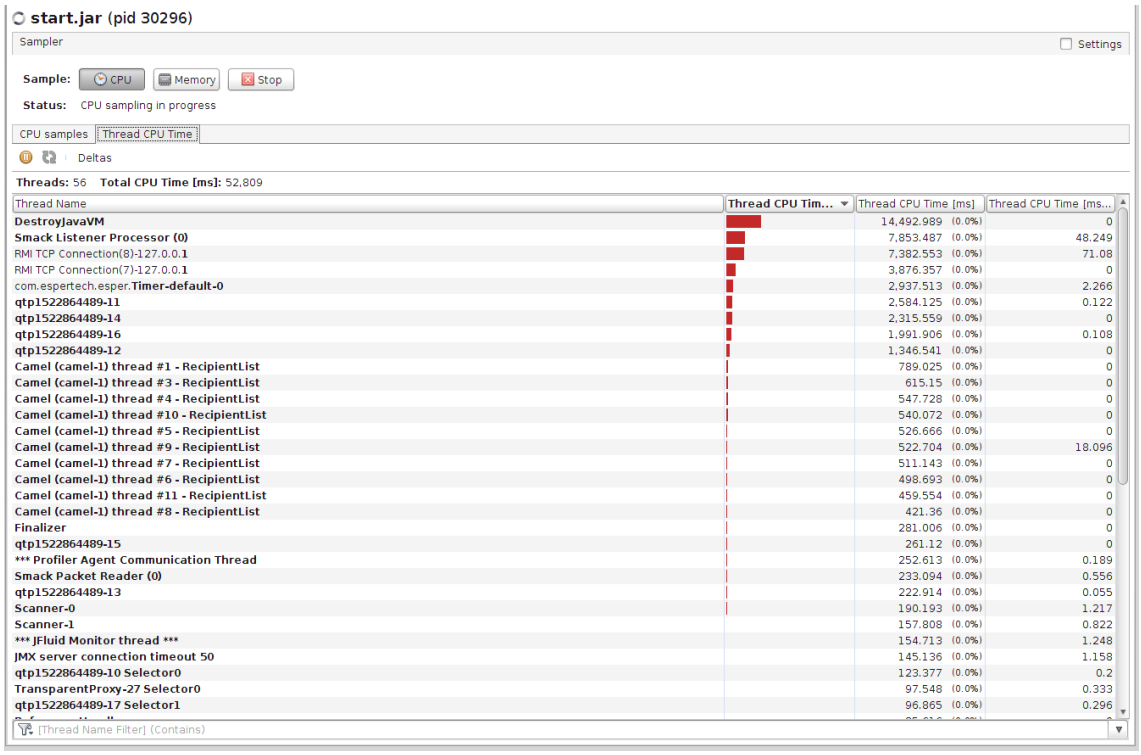

Figure A.55: CPU threads for the case of 60 messages per minute using Exp7

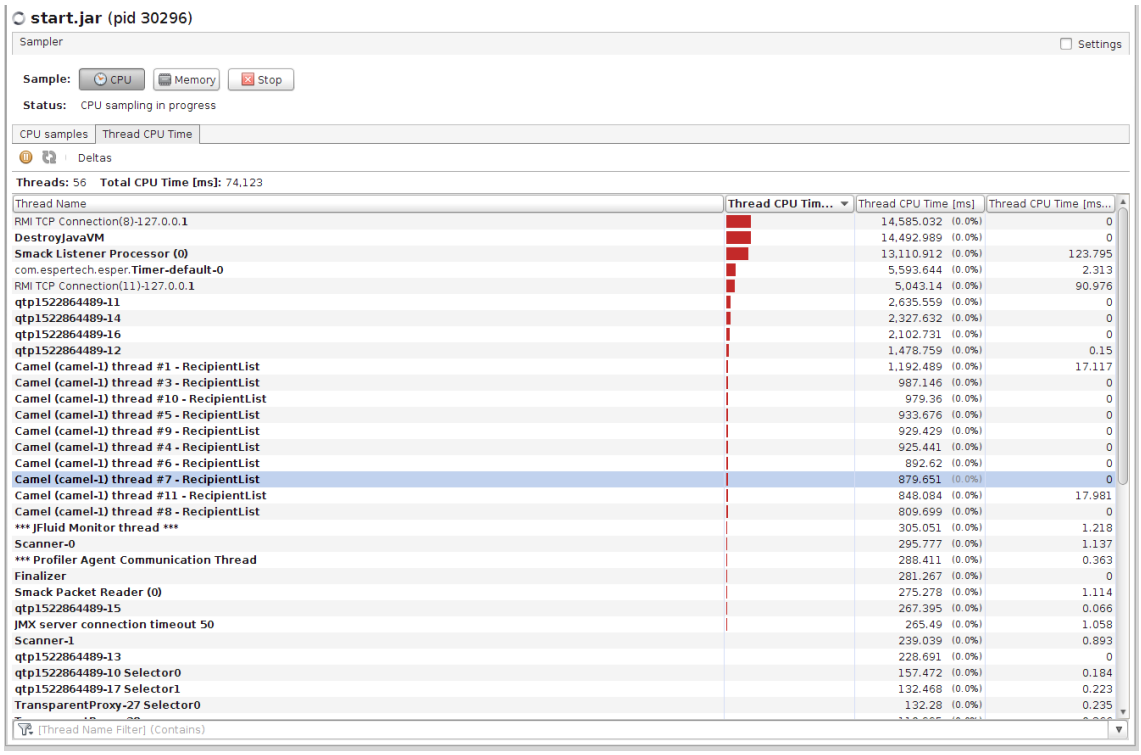

Figure A.56: CPU threads for the case of 120 messages per minute using Exp7

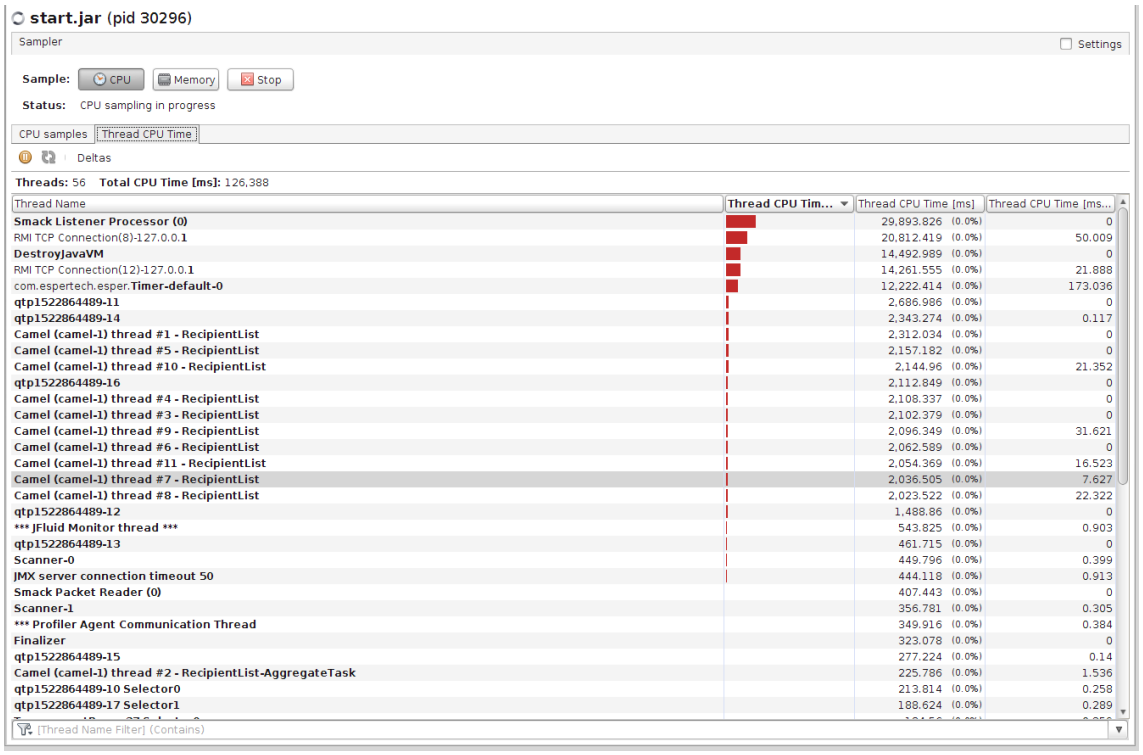

Figure A.57: CPU threads for the case of 240 messages per minute using Exp7

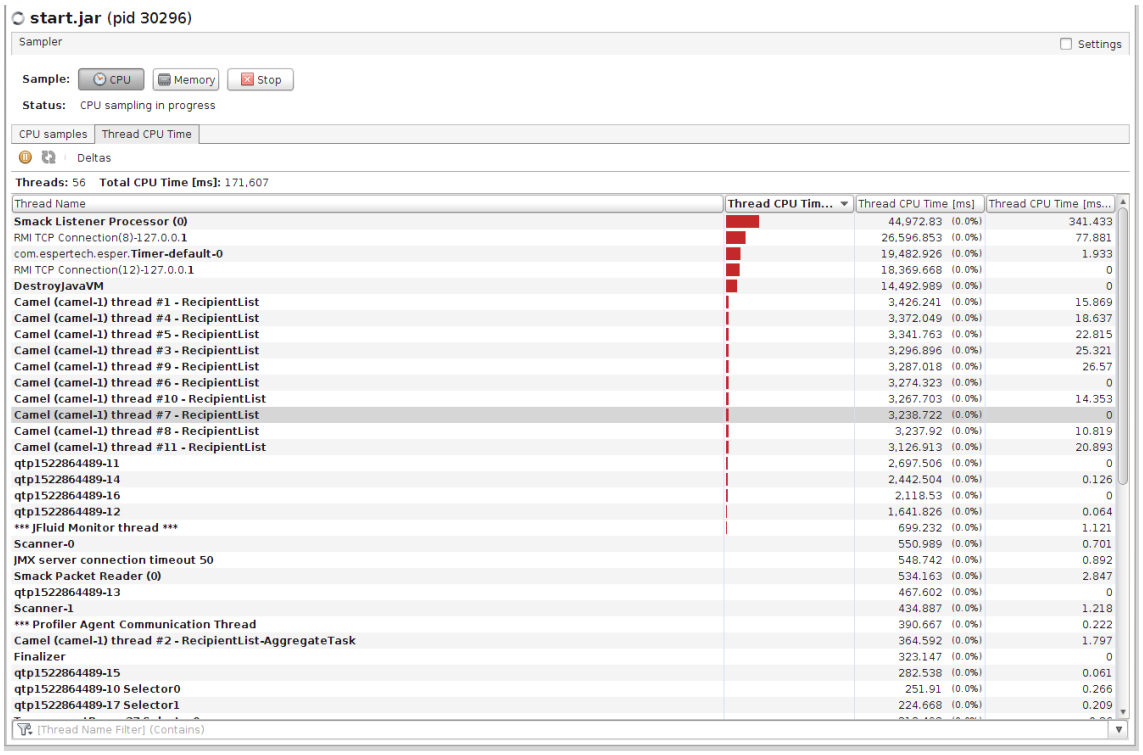

Figure A.58: CPU threads for the case of 480 messages per minute using Exp7

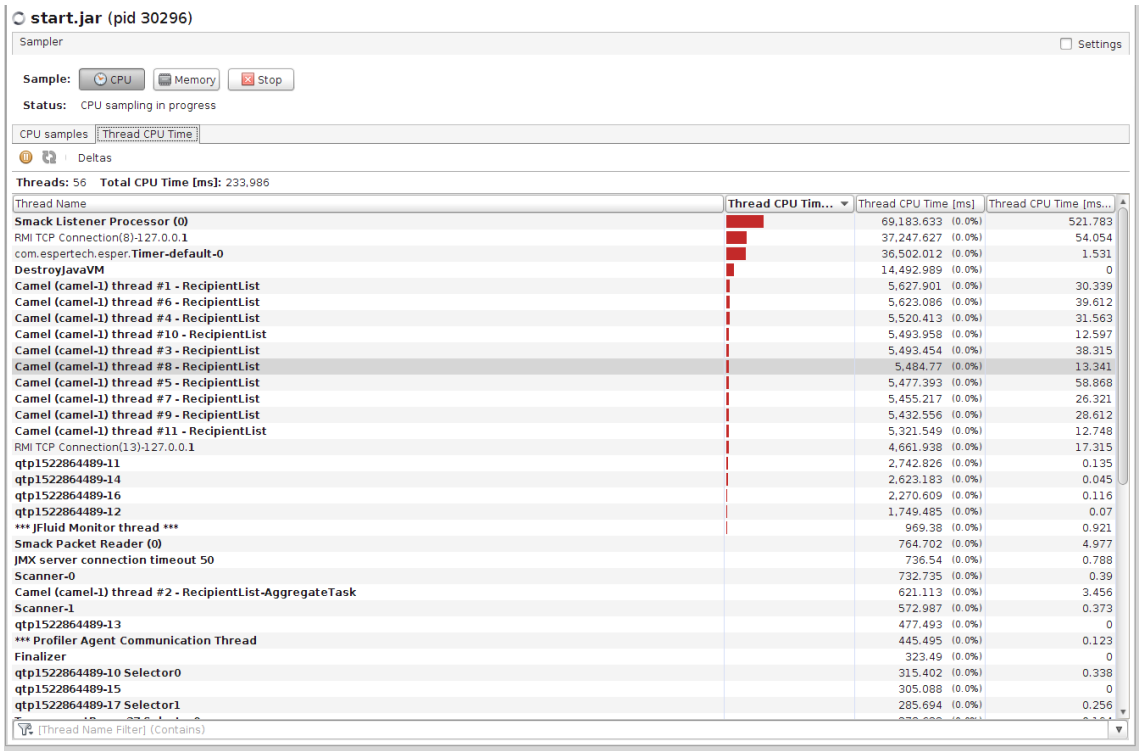

Figure A.59: CPU threads for the case of 1000 messages per minute using Exp7

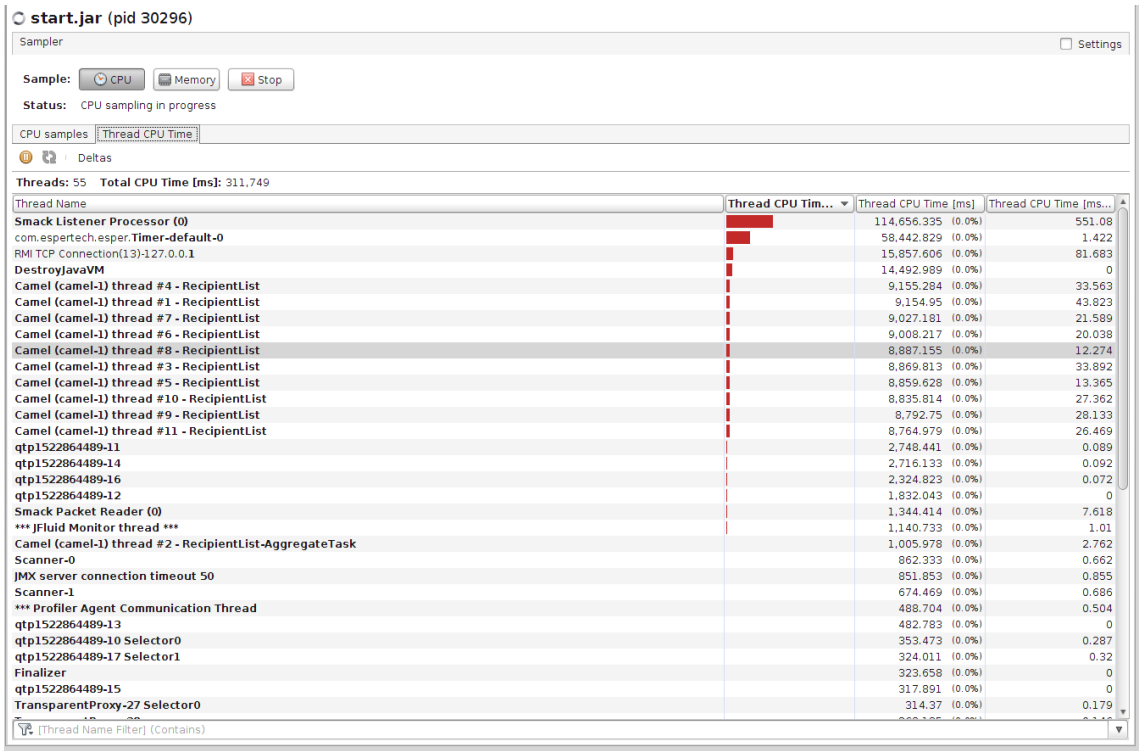

Figure A.60: CPU threads for the case of 2000 messages per minute using Exp7

A.1.5 Comparision graphs of middleware delay

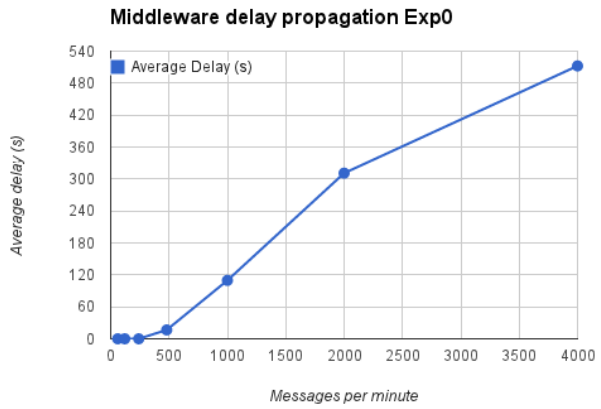

Figure A.61: Middleware delay propagation using Exp0

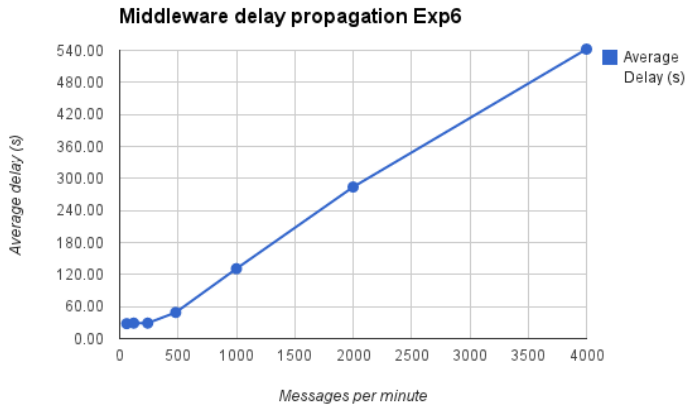

Figure A.62: Middleware delay propagation using Exp6

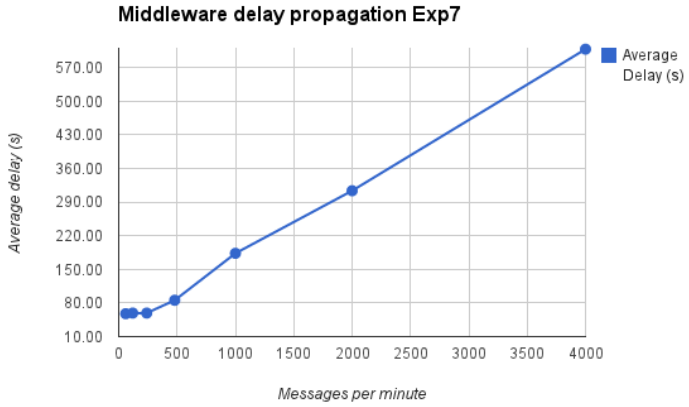

Figure A.63: Middleware delay propagation using Exp7

A.2 1 source, 2 sessions, 1 client full info

A.2.1 visualVM monitor screenshots

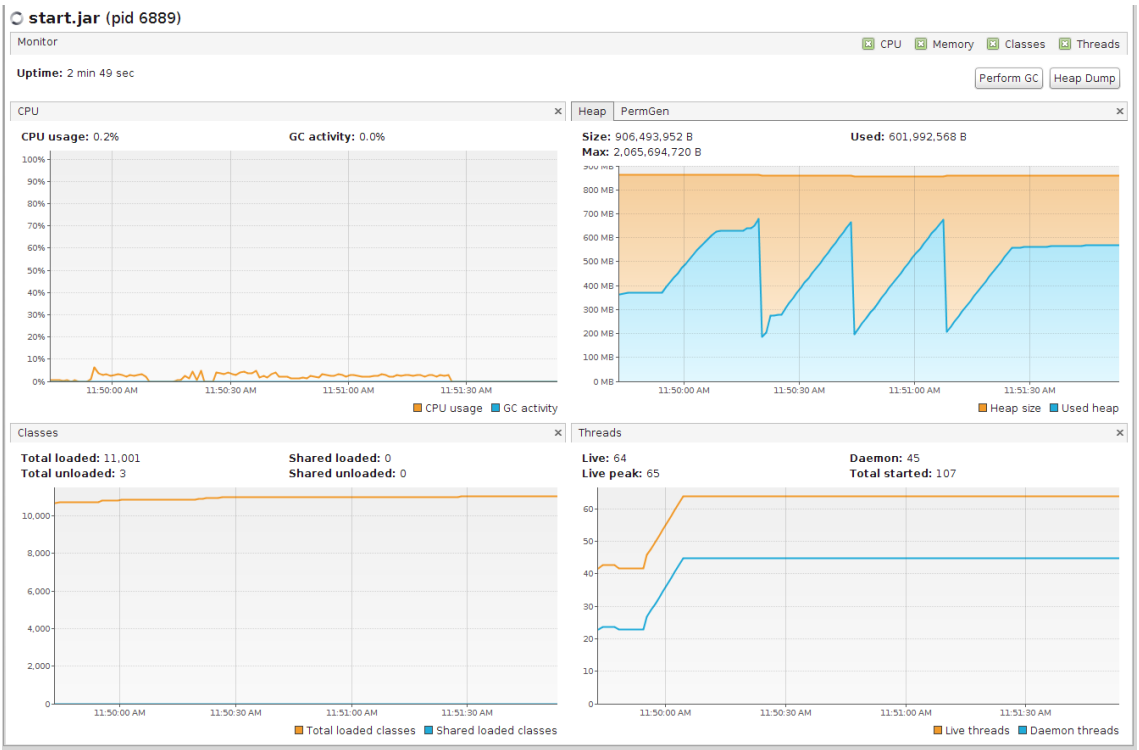

Figure A.64: Resources used when running Exp0 with 60 messages per minute

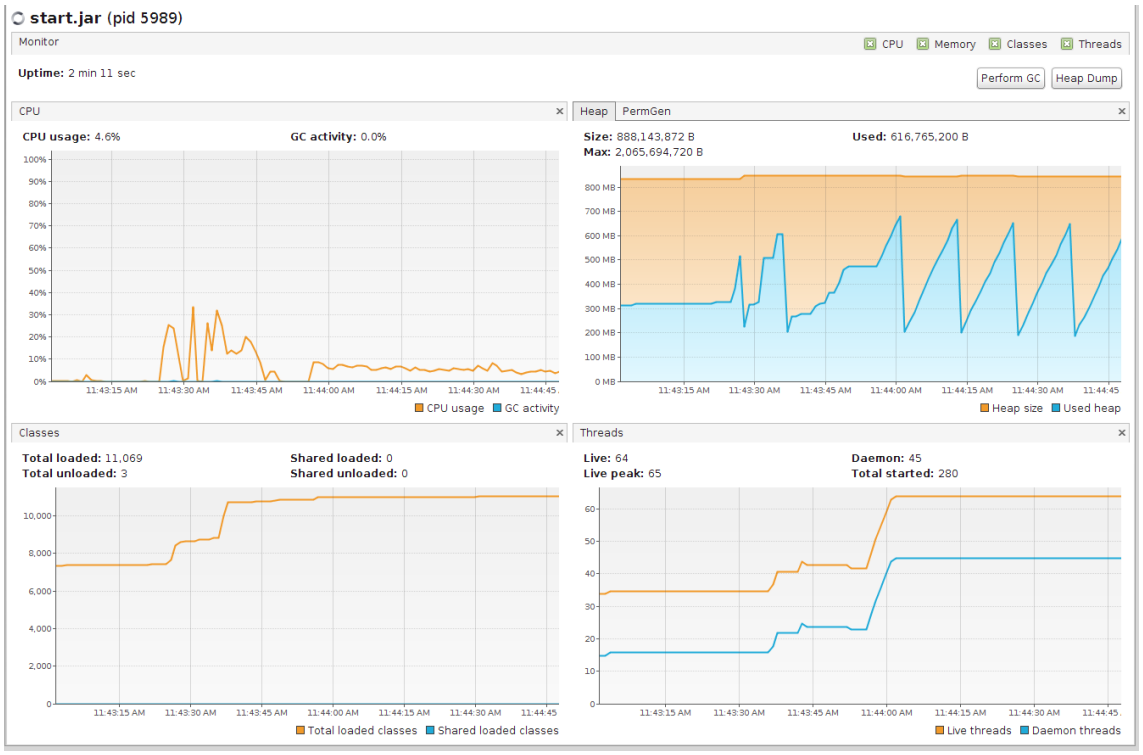

Figure A.65: Resources used when running Exp0 with 120 messages per minute

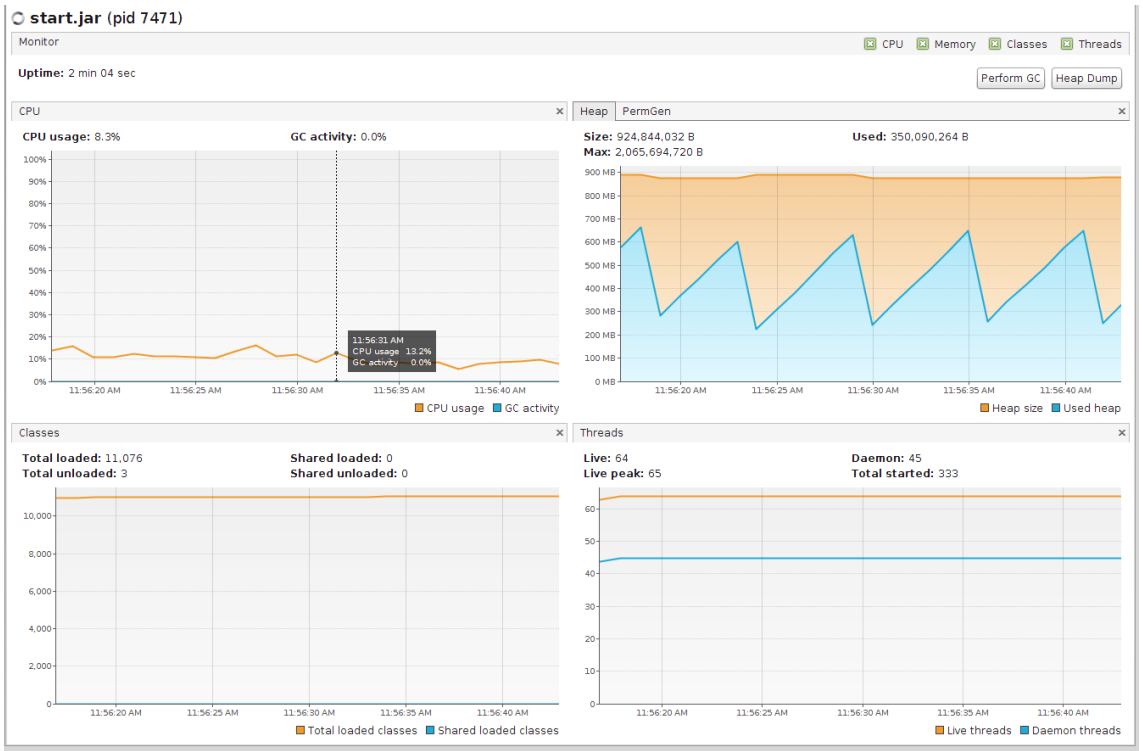

Figure A.66: Resources used when running Exp0 with 240 messages per minute

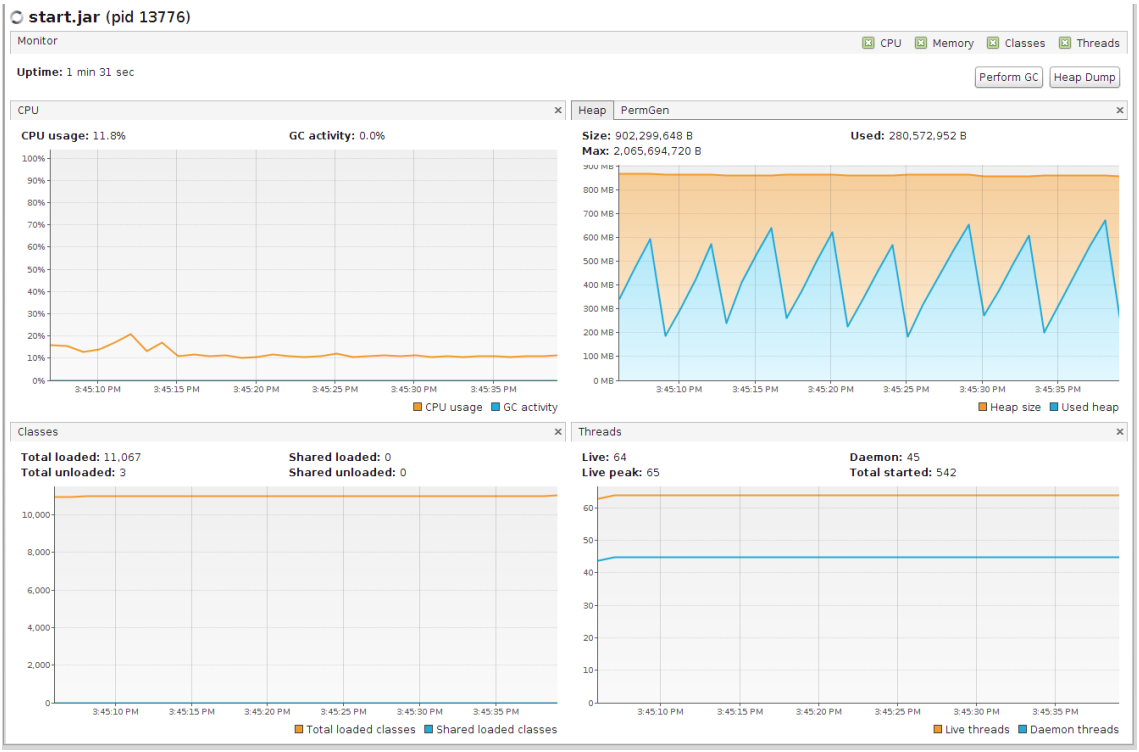

Figure A.67: Resources used when running Exp0 with 480 messages per minute

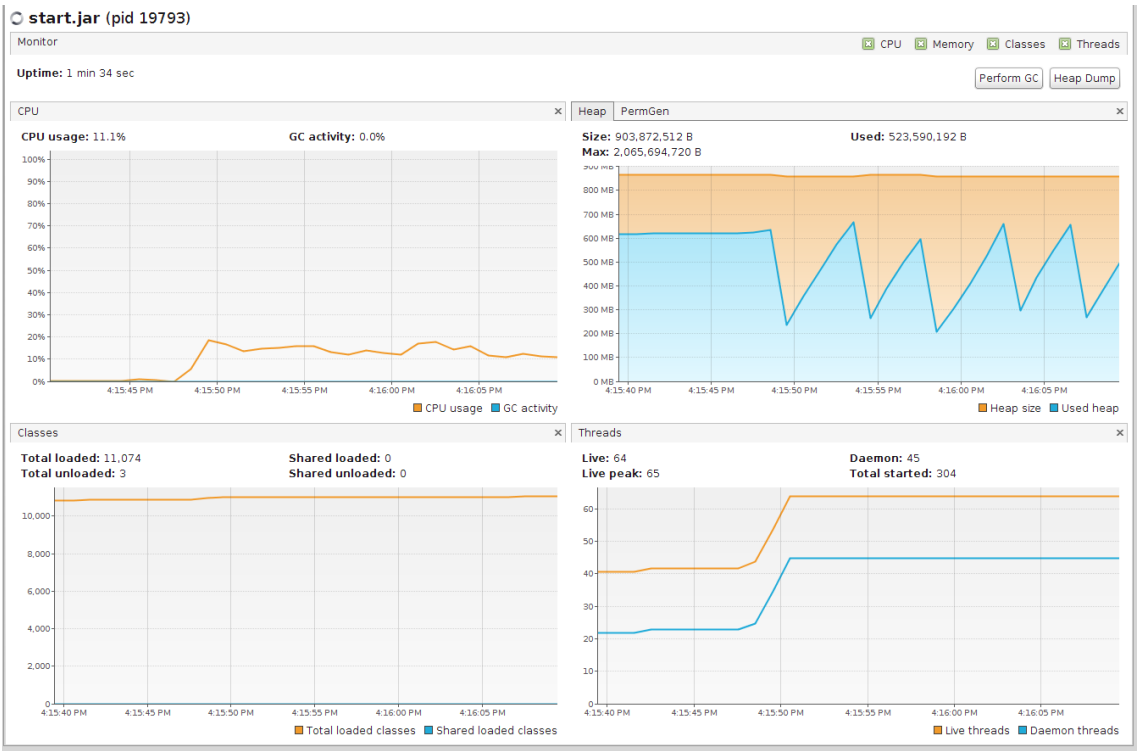

Figure A.68: Resources used when running Exp0 with 1000 messages per minute

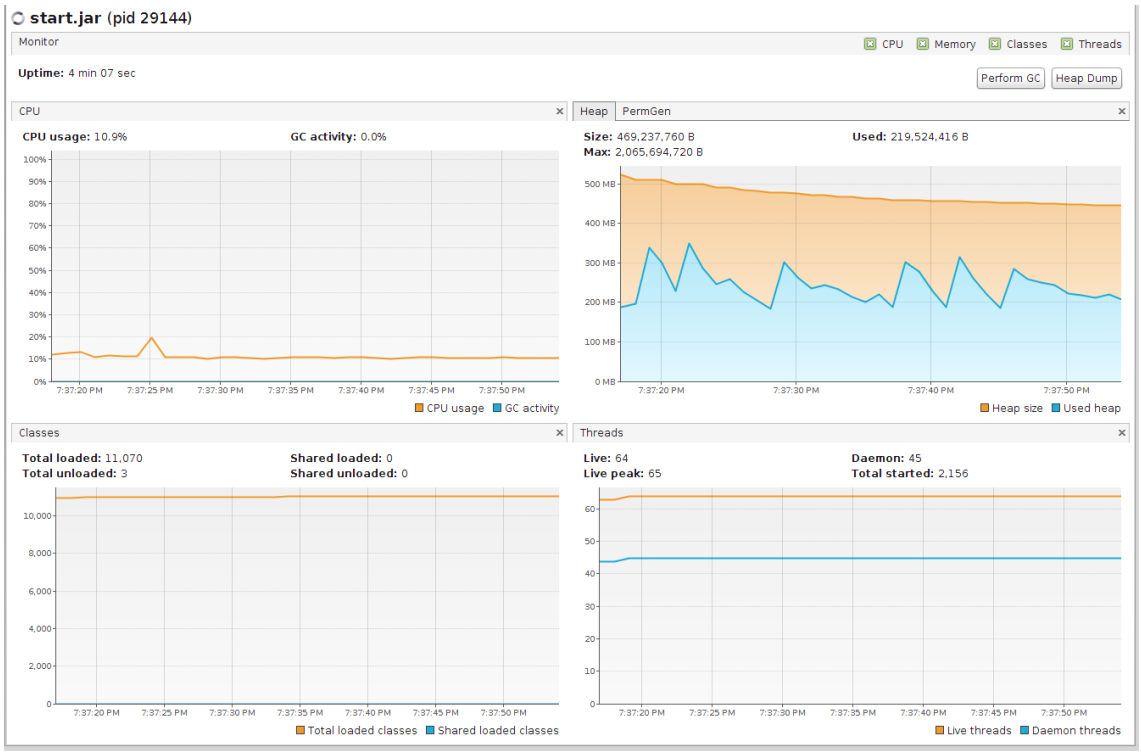

Figure A.69: Resources used when running Exp0 with 2000 messages per minute

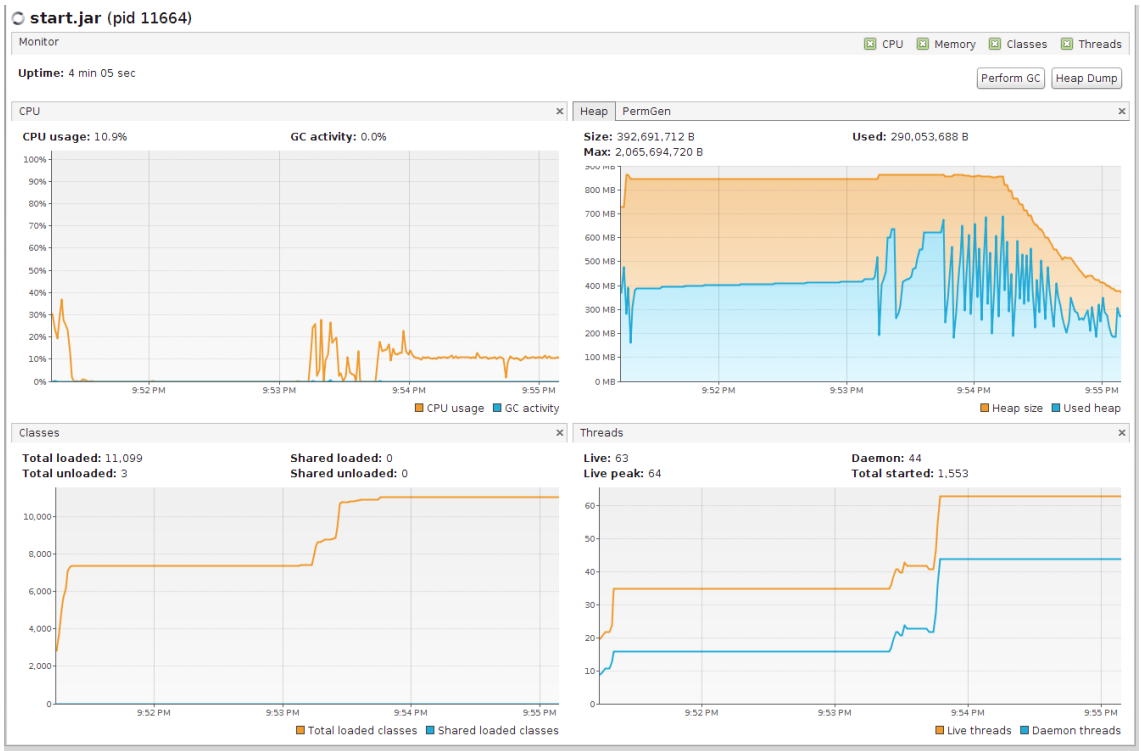

Figure A.70: Resources used when running Exp0 with 4000 messages per minute

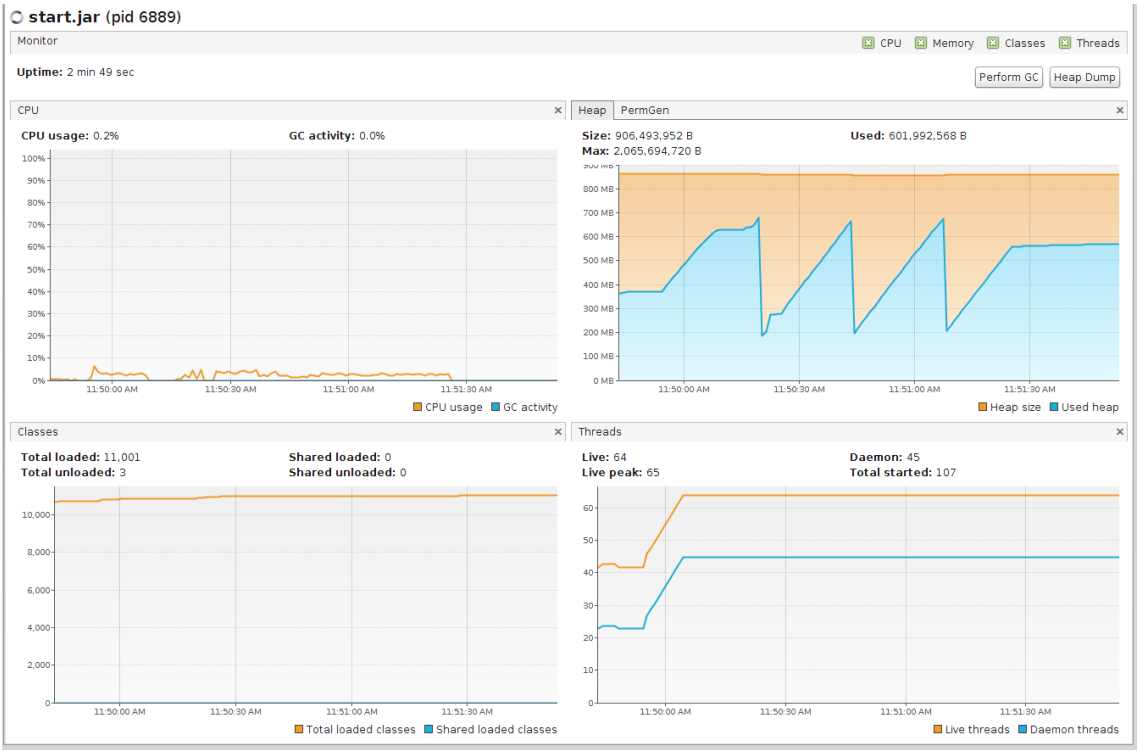

Figure A.71: Resources used when running Exp6 with 60 messages per minute

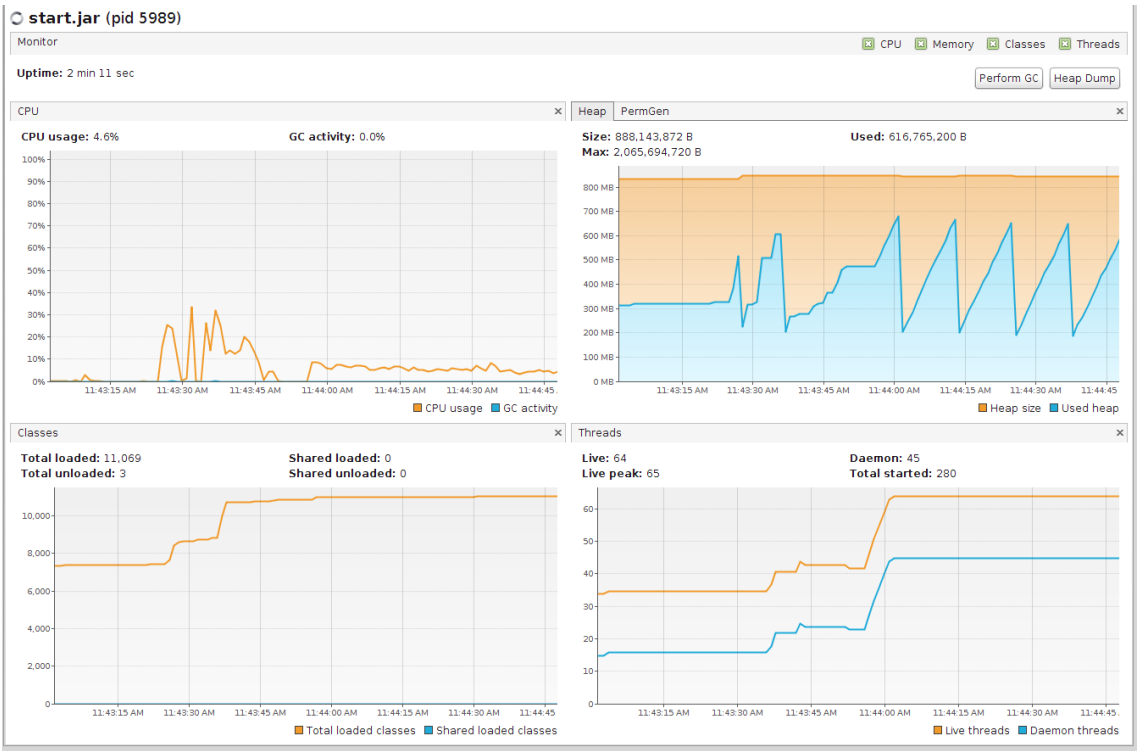

Figure A.72: Resources used when running Exp6 with 120 messages per minute

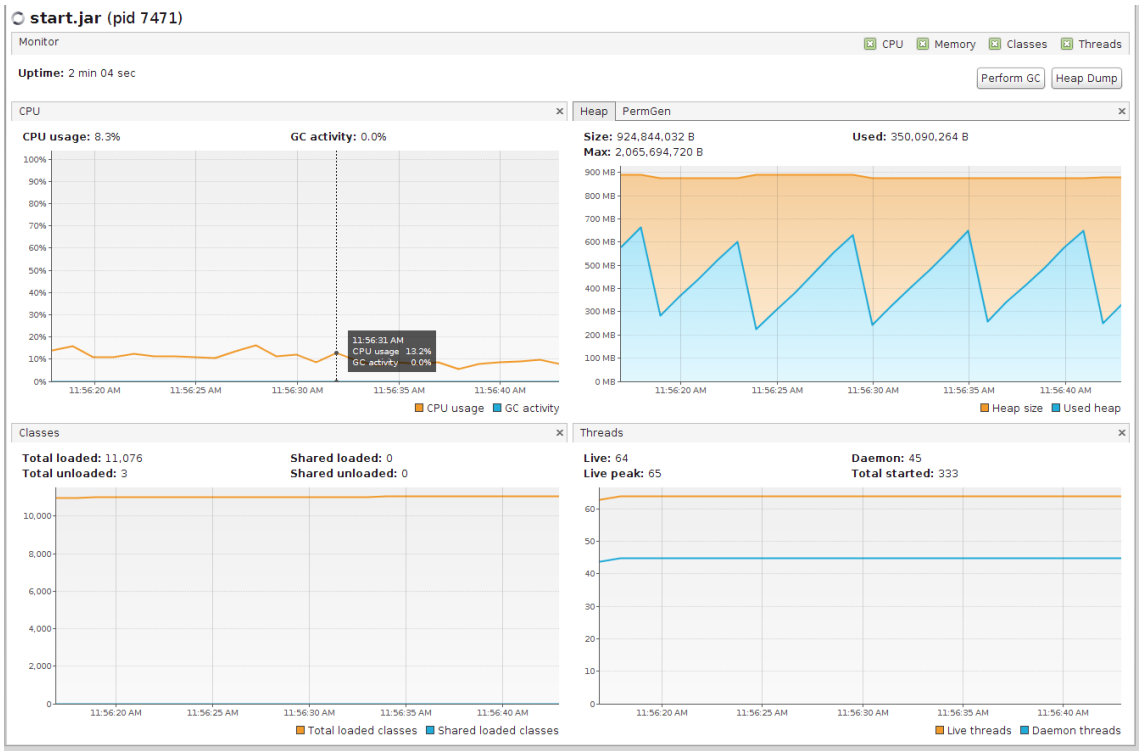

Figure A.73: Resources used when running Exp6 with 240 messages per minute

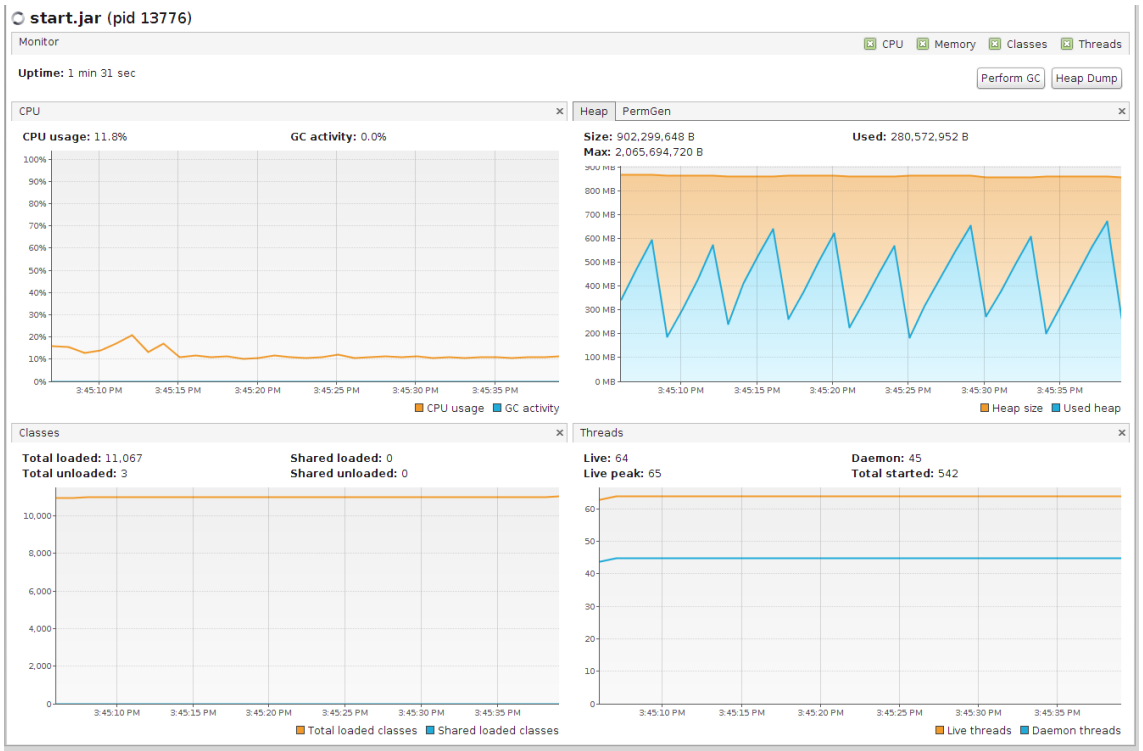

Figure A.74: Resources used when running Exp6 with 480 messages per minute

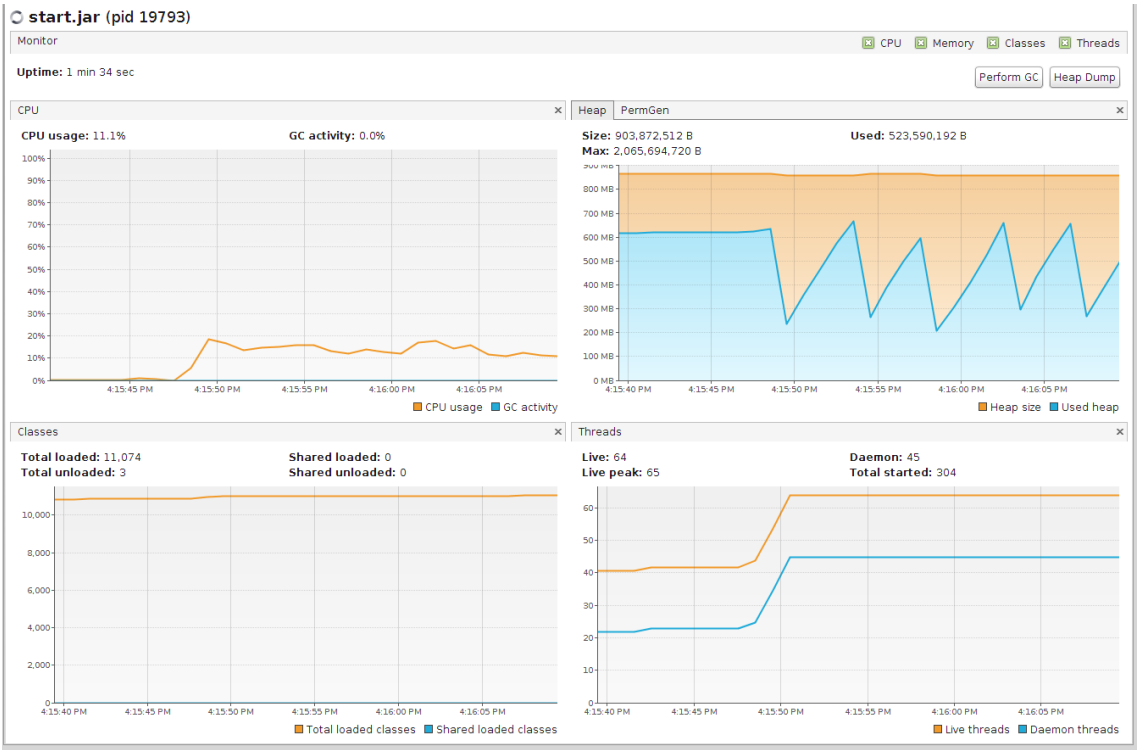

Figure A.75: Resources used when running Exp6 with 1000 messages per minute

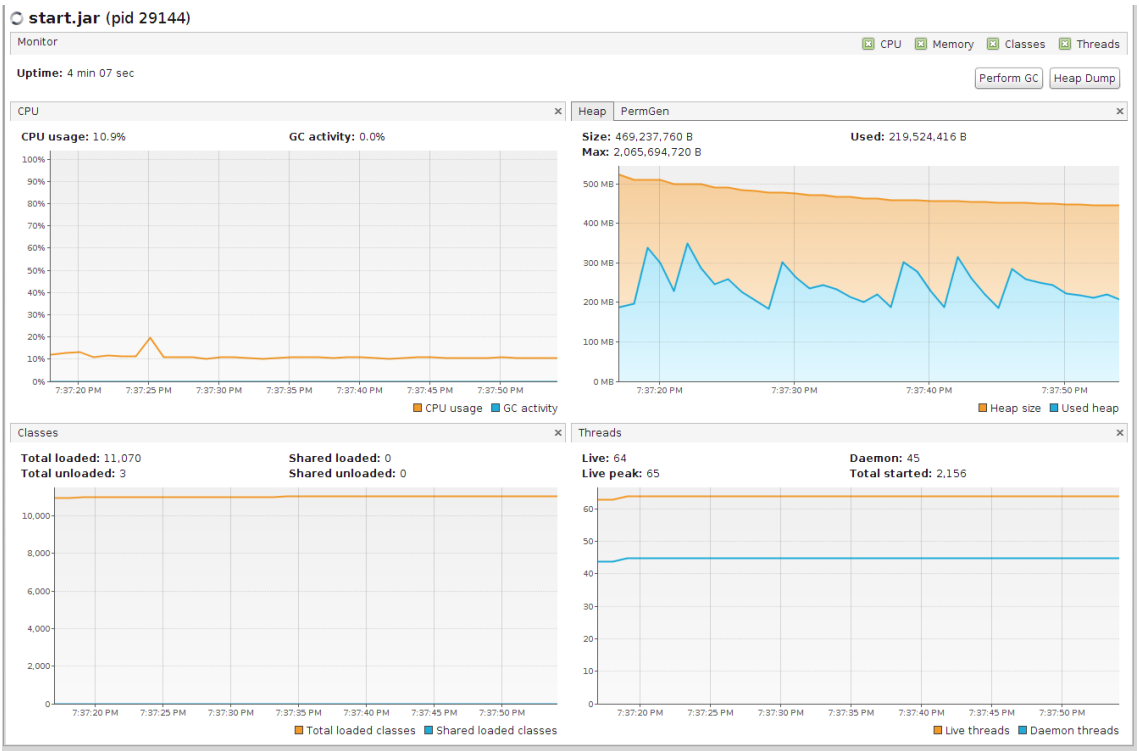

Figure A.76: Resources used when running Exp6 with 2000 messages per minute

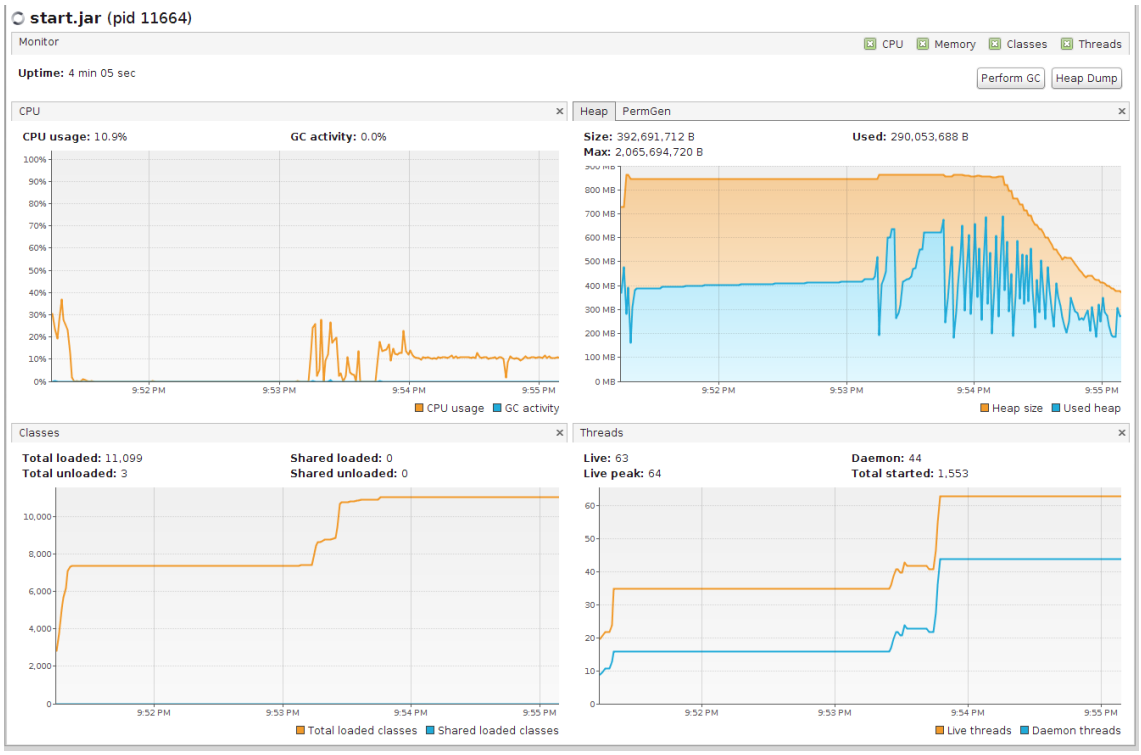

Figure A.77: Resources used when running Exp6 with 4000 messages per minute

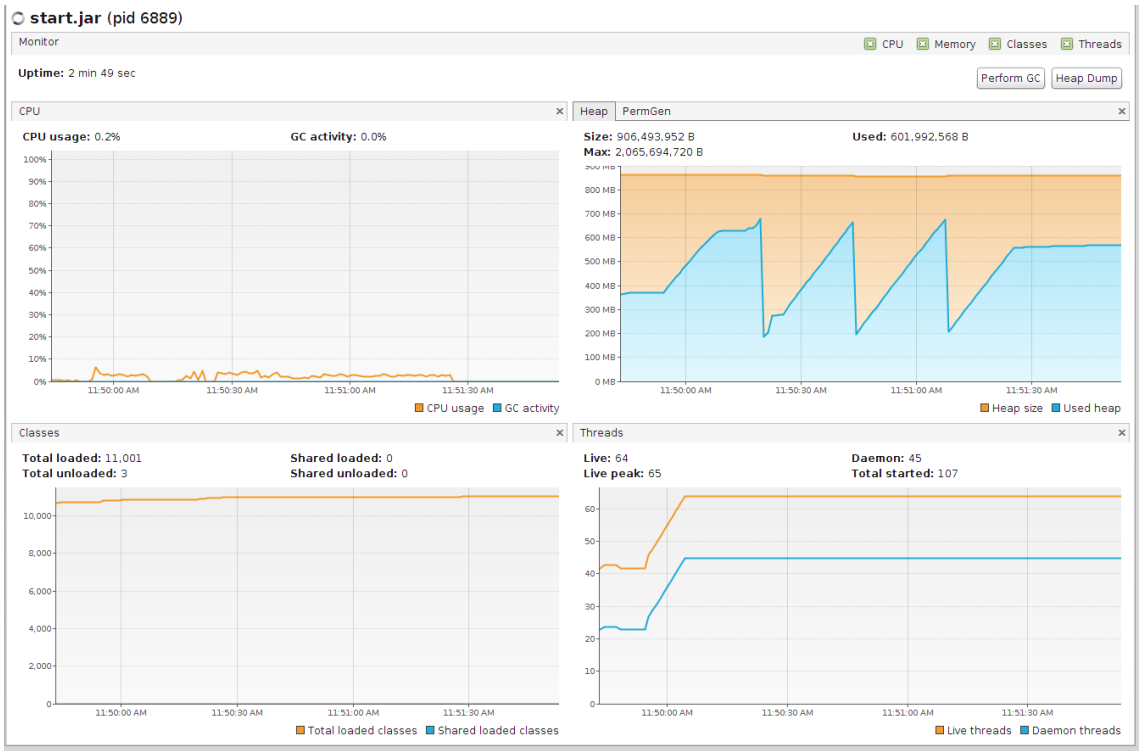

Figure A.78: Resources used when running Exp7 with 60 messages per minute

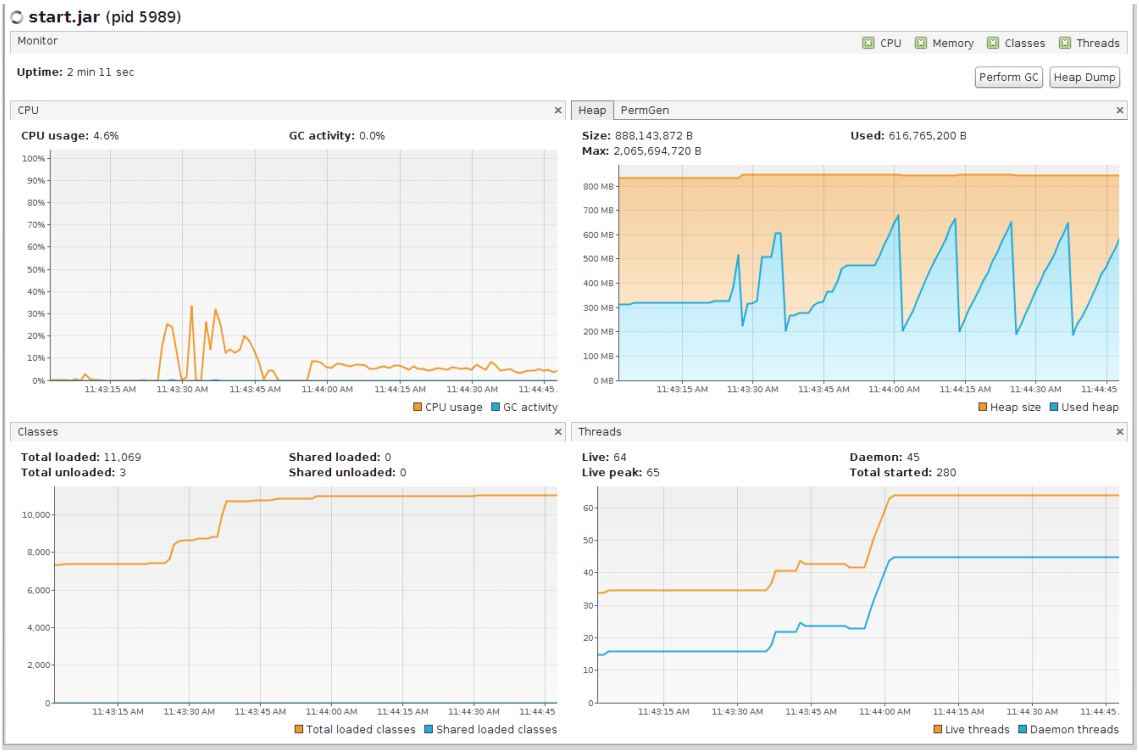

Figure A.79: Resources used when running Exp7 with 120 messages per minute

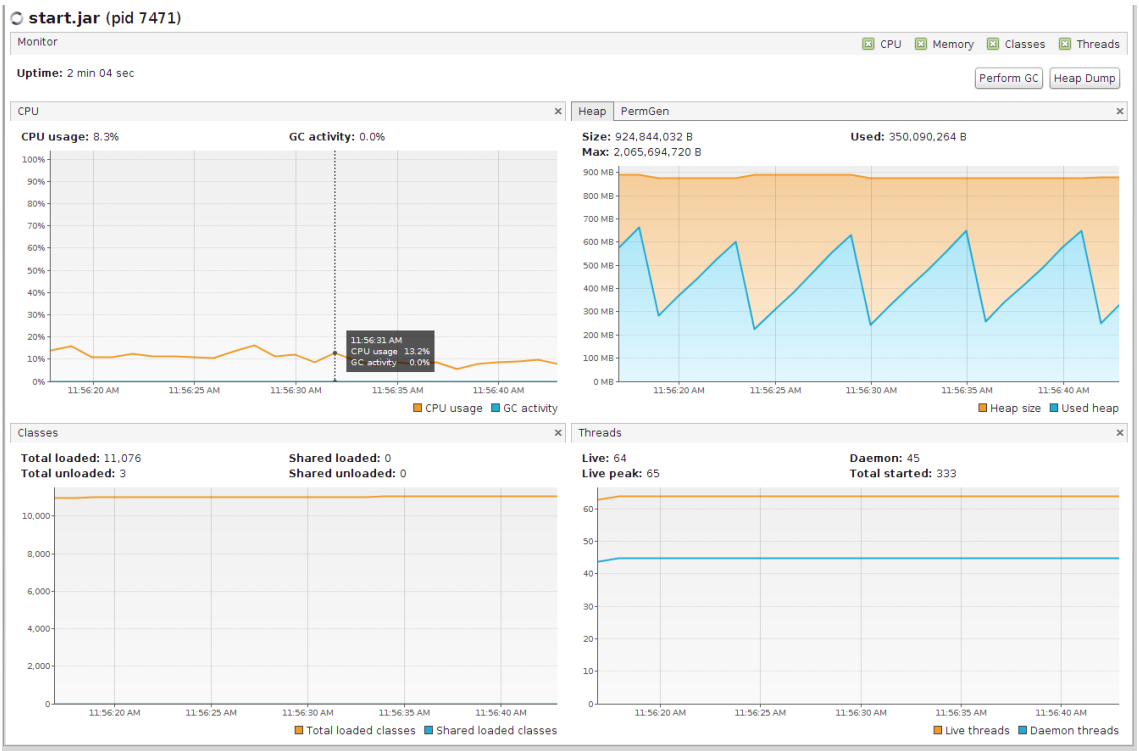

Figure A.80: Resources used when running Exp7 with 240 messages per minute

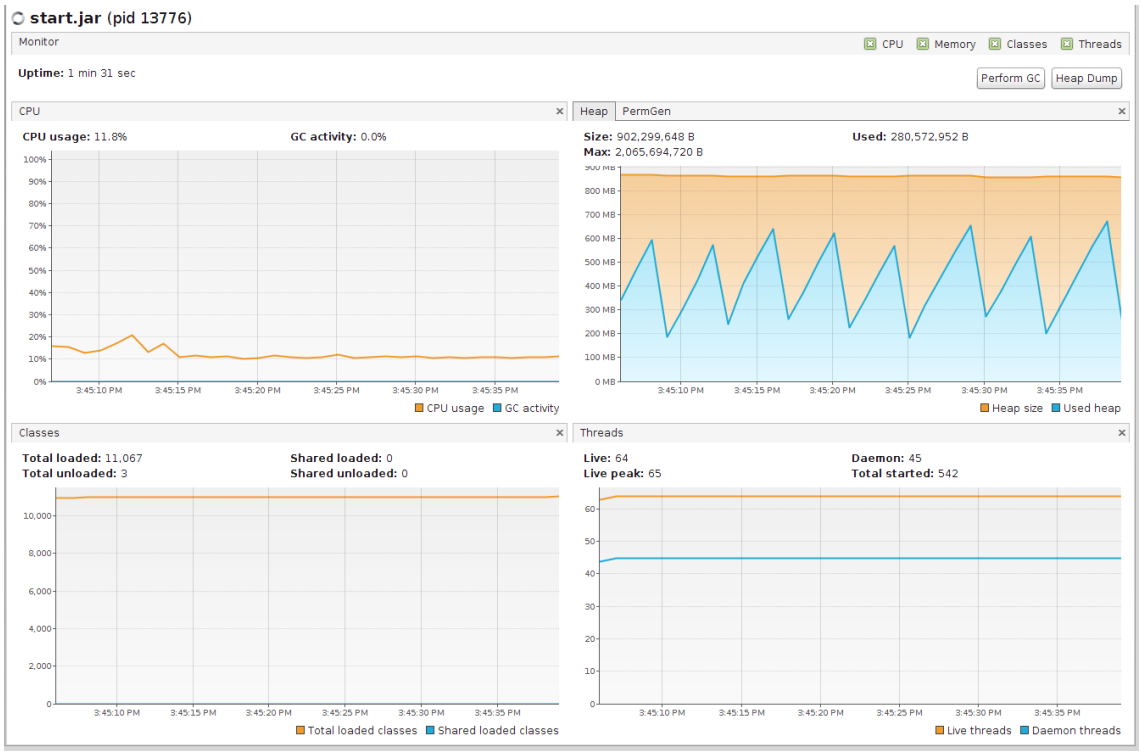

Figure A.81: Resources used when running Exp7 with 480 messages per minute

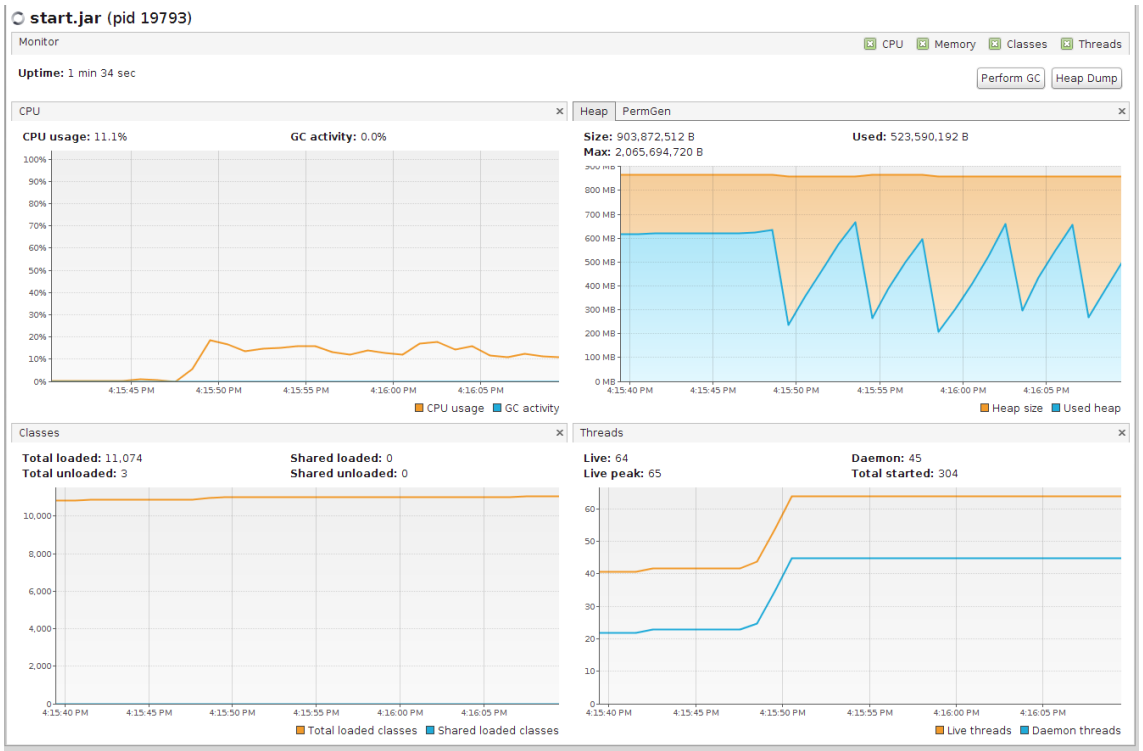

Figure A.82: Resources used when running Exp7 with 1000 messages per minute

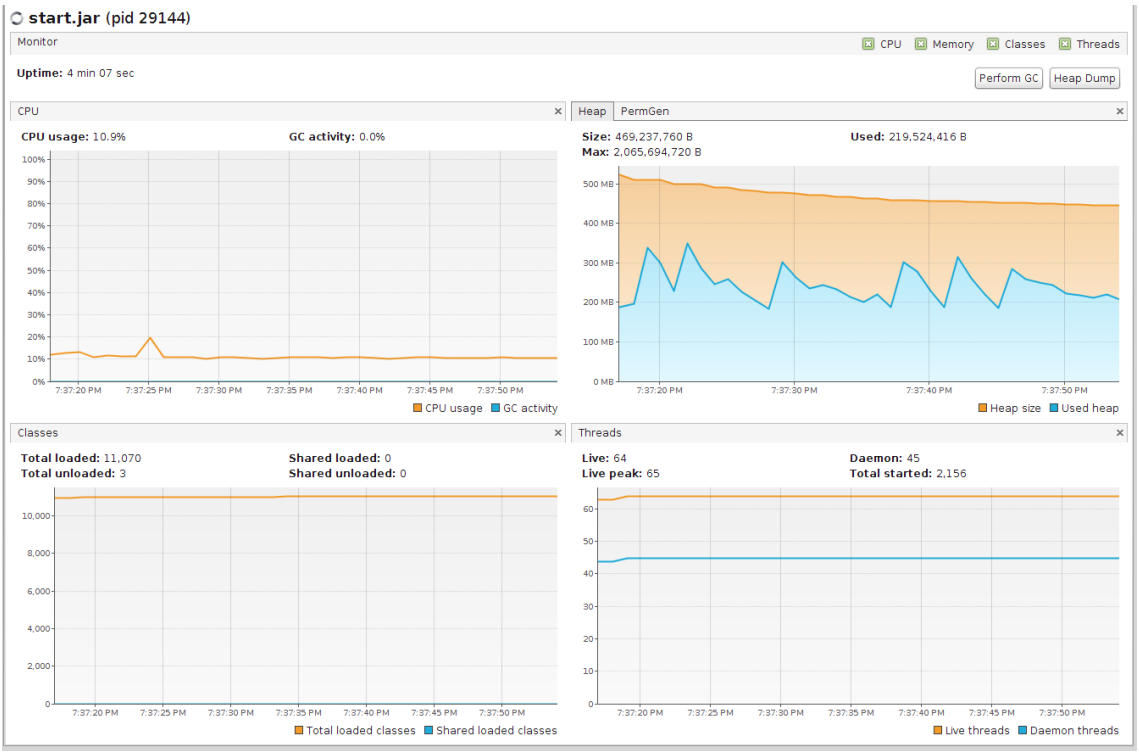

Figure A.83: Resources used when running Exp7 with 2000 messages per minute

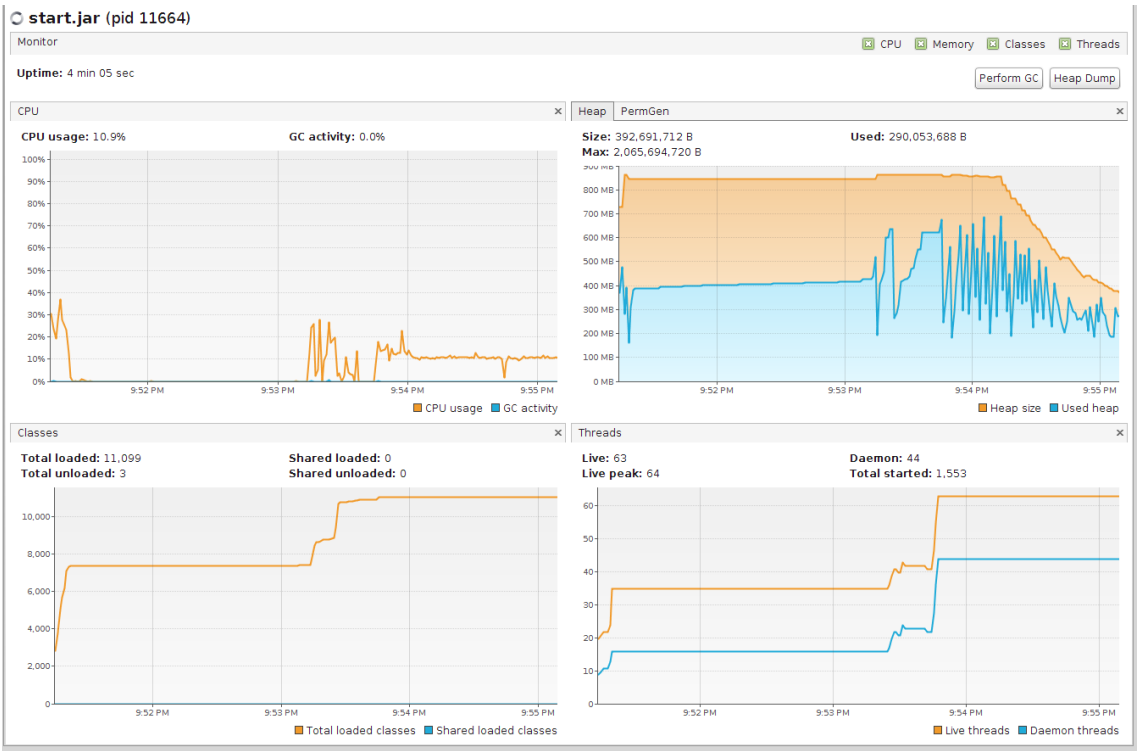

Figure A.84: Resources used when running Exp7 with 4000 messages per minute

A.2.2 Comparision graphs of middleware delay

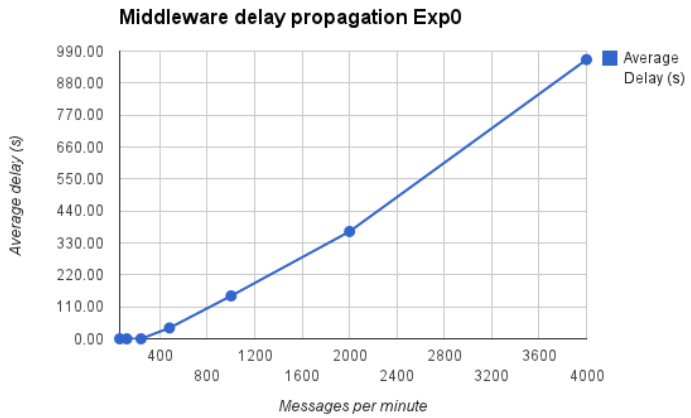

Figure A.85: Middleware delay propagation using Exp0

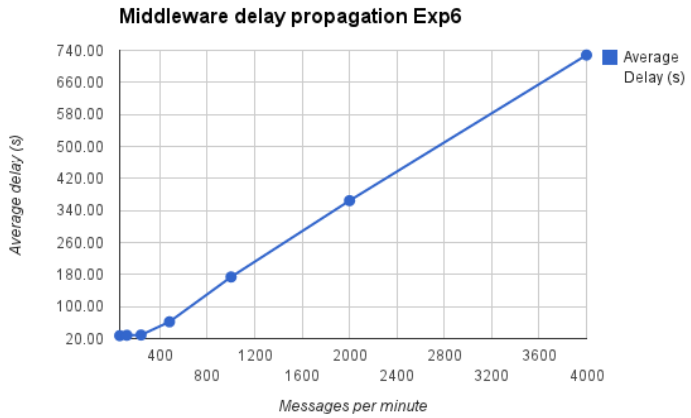

Figure A.86: Middleware delay propagation using Exp6

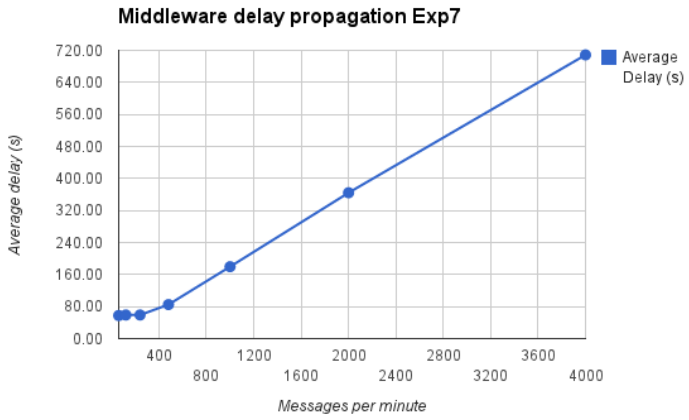

Figure A.87: Middleware delay propagation using Exp7

A.3 1 source, 4 sessions, 1 client full info

A.3.1 visualVM monitor screenshots

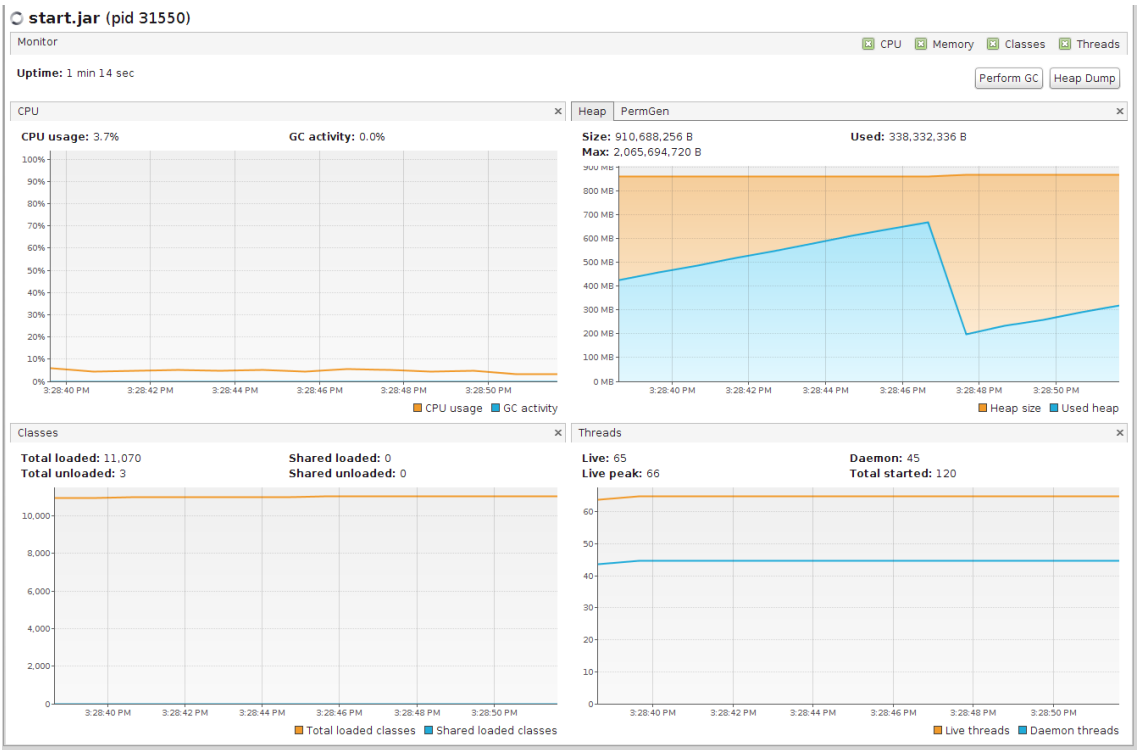

Figure A.88: Resources used when running Exp0 with 60 messages per minute

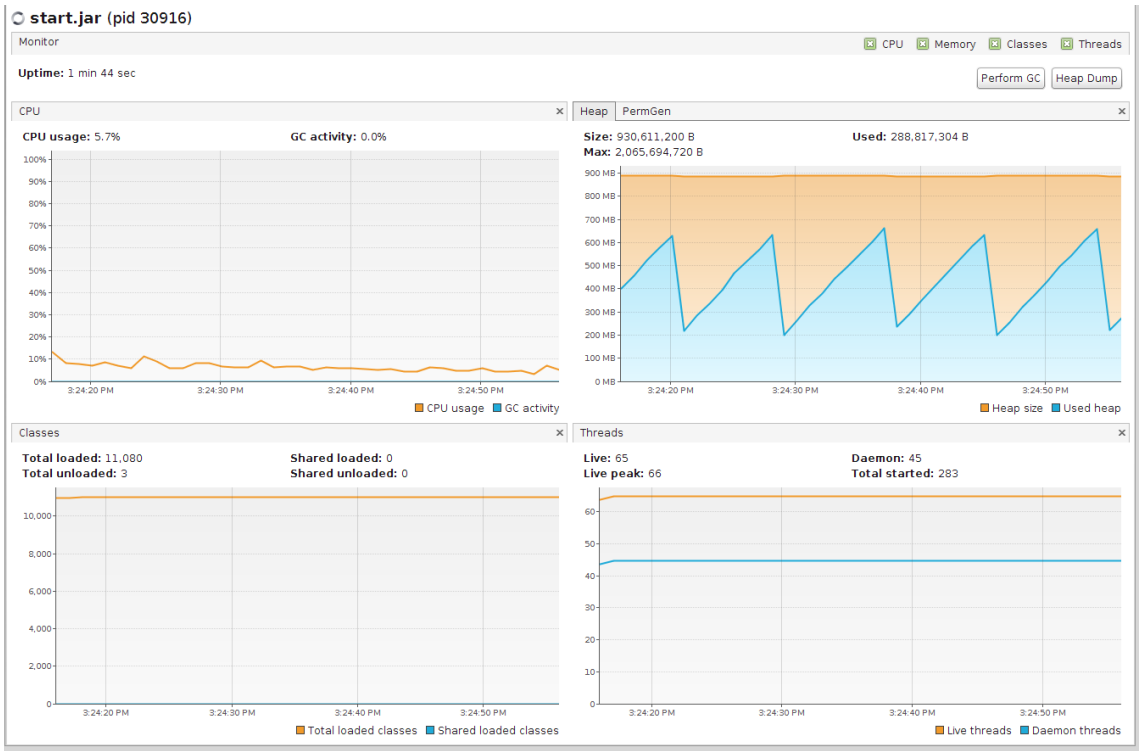

Figure A.89: Resources used when running Exp0 with 120 messages per minute

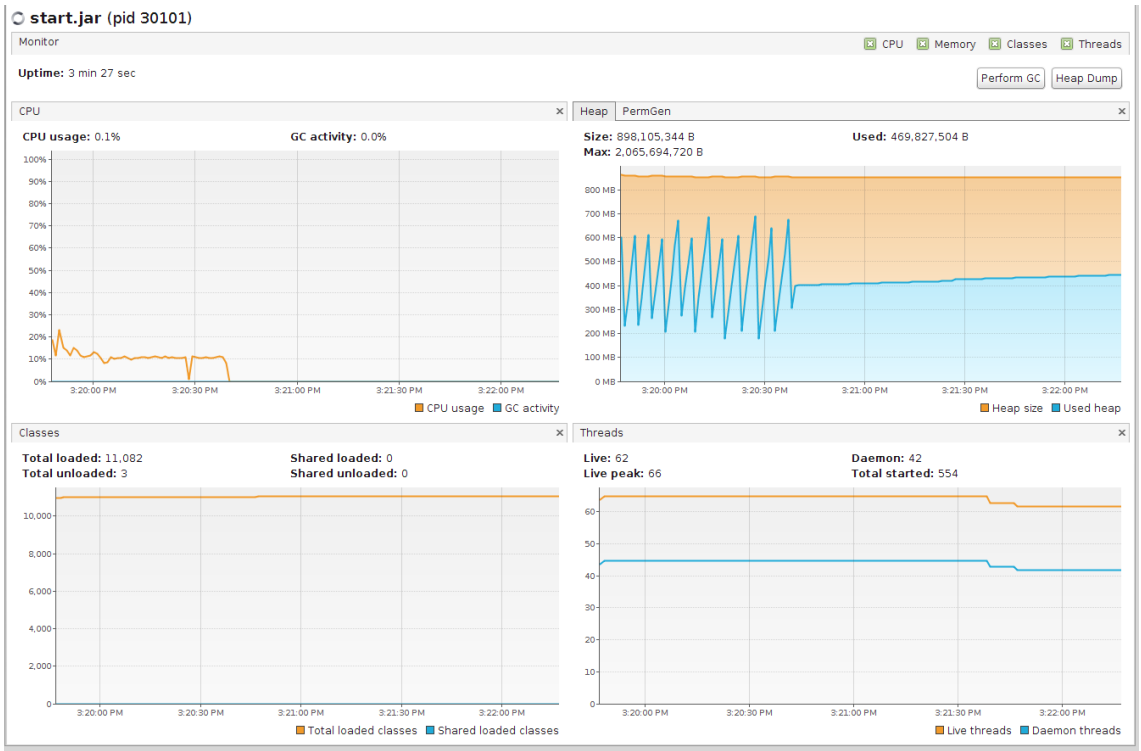

Figure A.90: Resources used when running Exp0 with 240 messages per minute

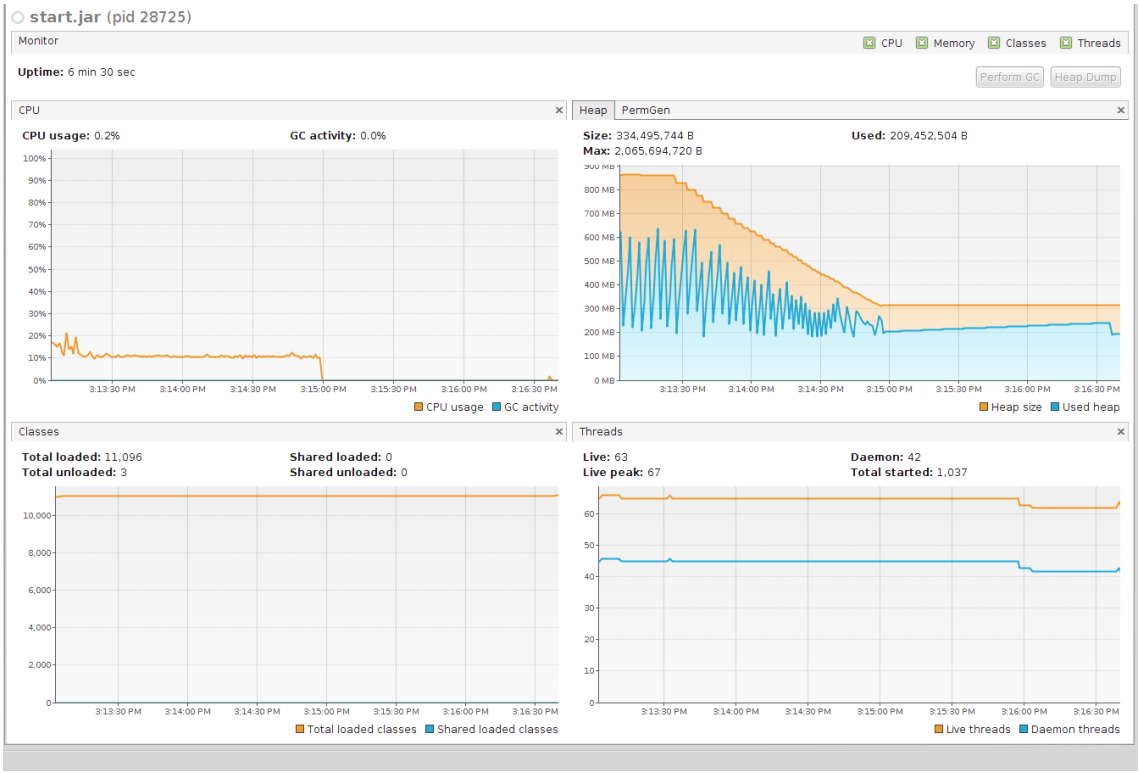

Figure A.91: Resources used when running Exp0 with 480 messages per minute

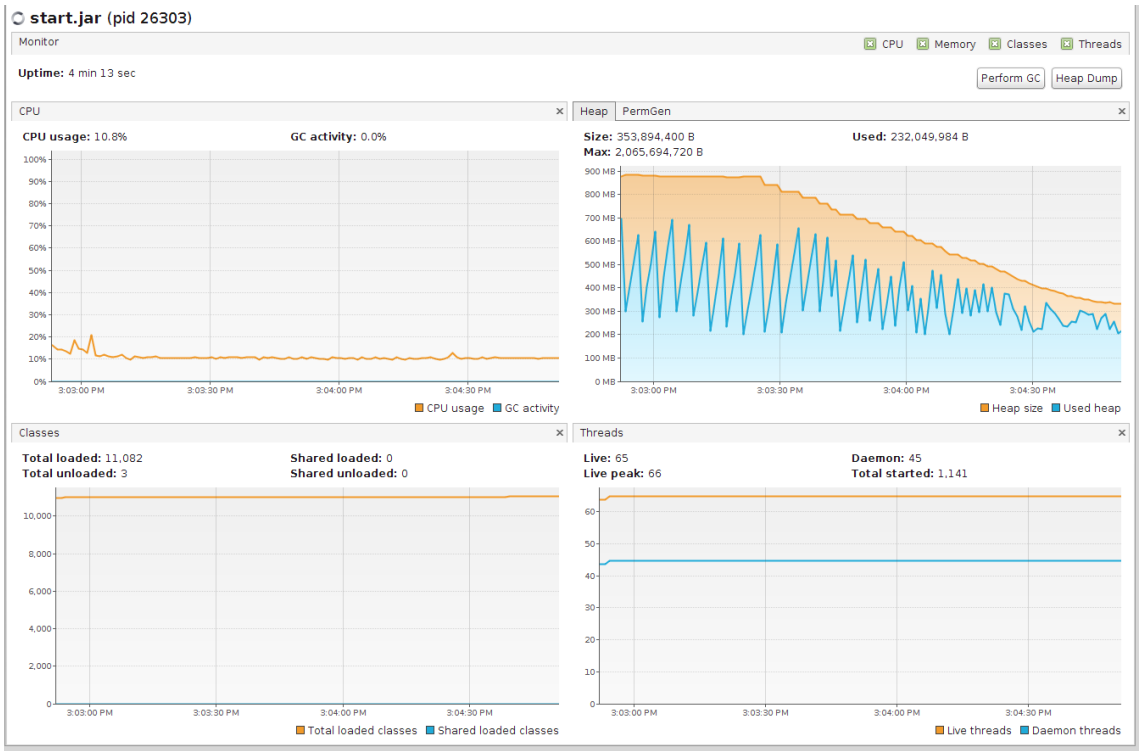

Figure A.92: Resources used when running Exp0 with 1000 messages per minute

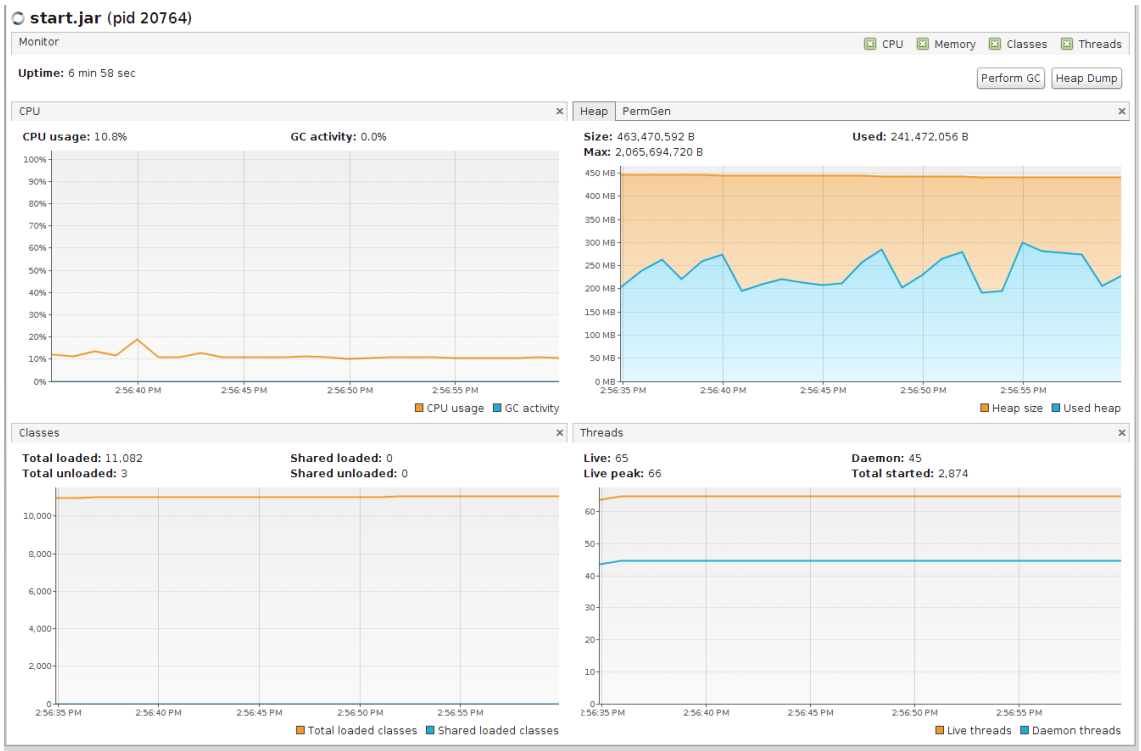

Figure A.93: Resources used when running Exp0 with 2000 messages per minute

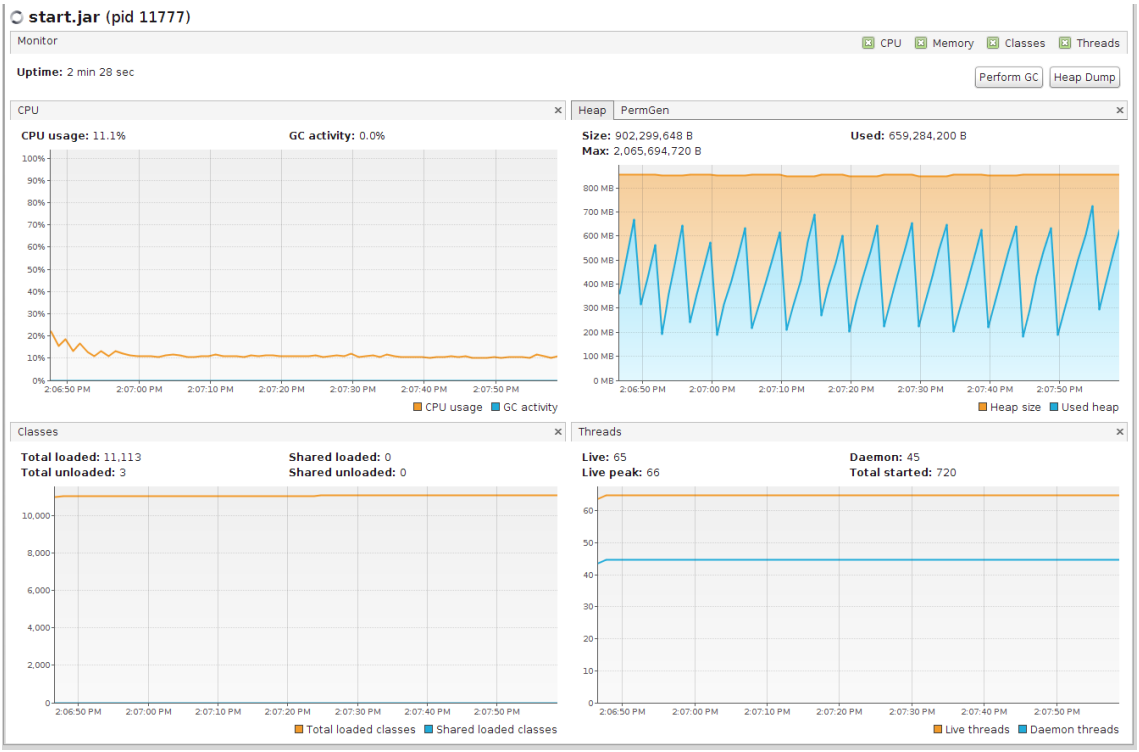

Figure A.94: Resources used when running Exp0 with 4000 messages per minute

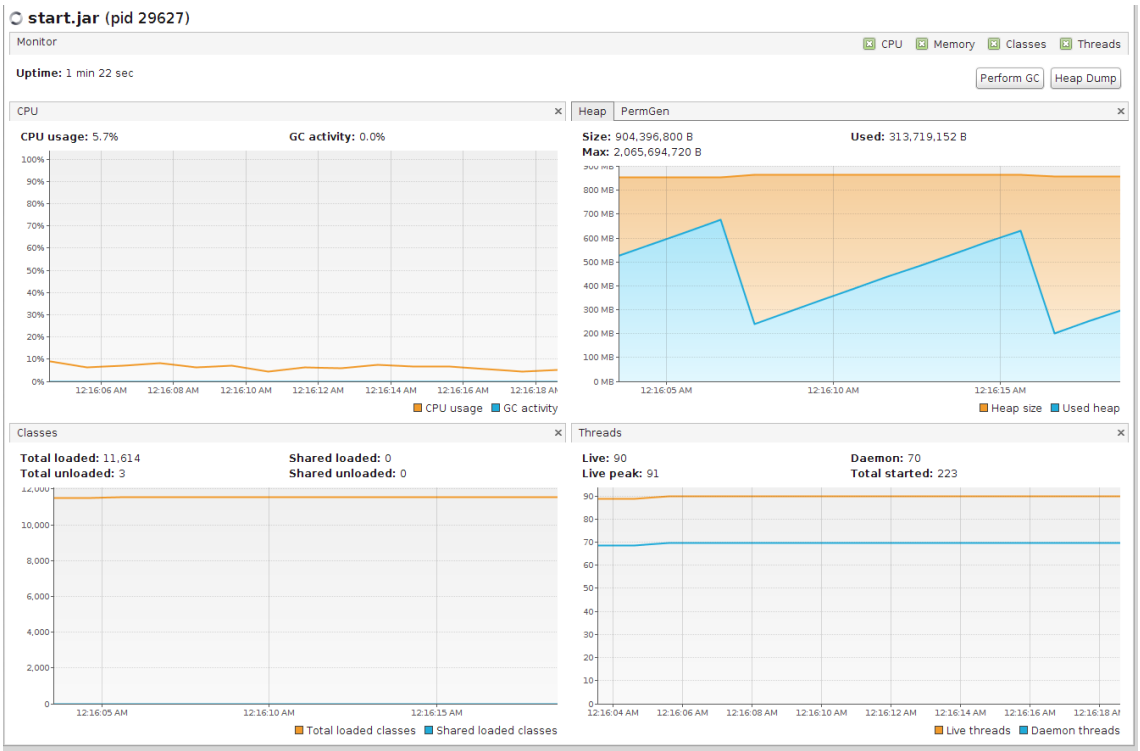

Figure A.95: Resources used when running Exp6 with 60 messages per minute

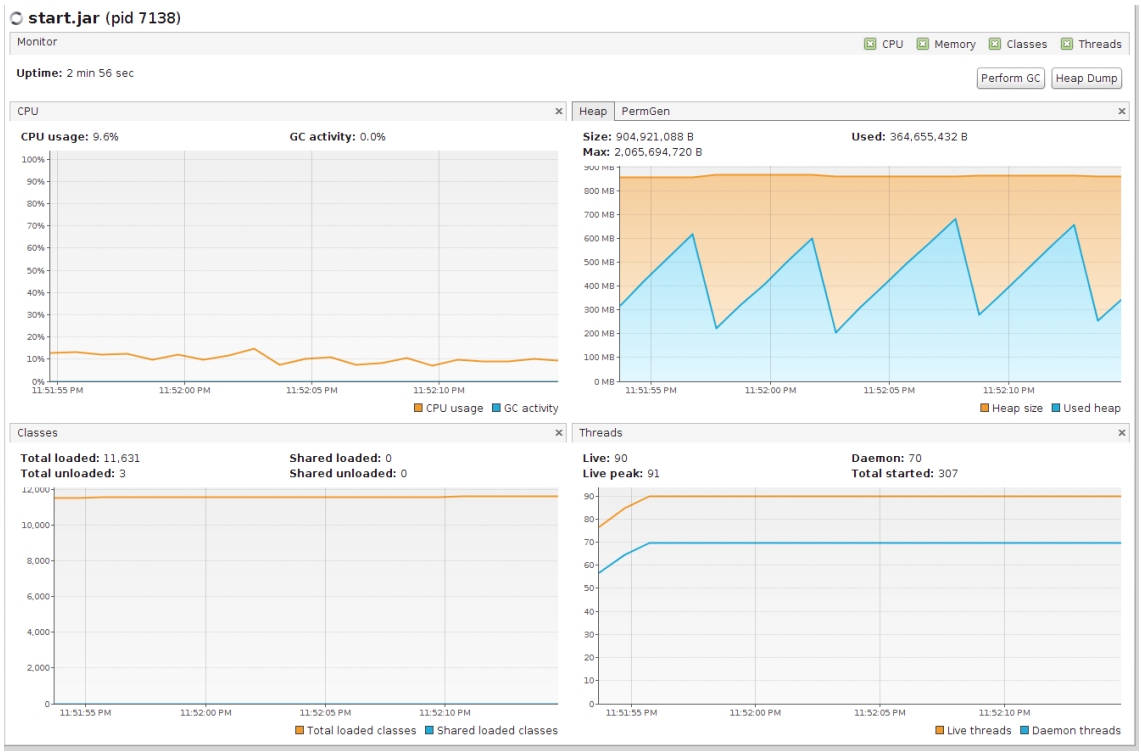

Figure A.96: Resources used when running Exp6 with 120 messages per minute

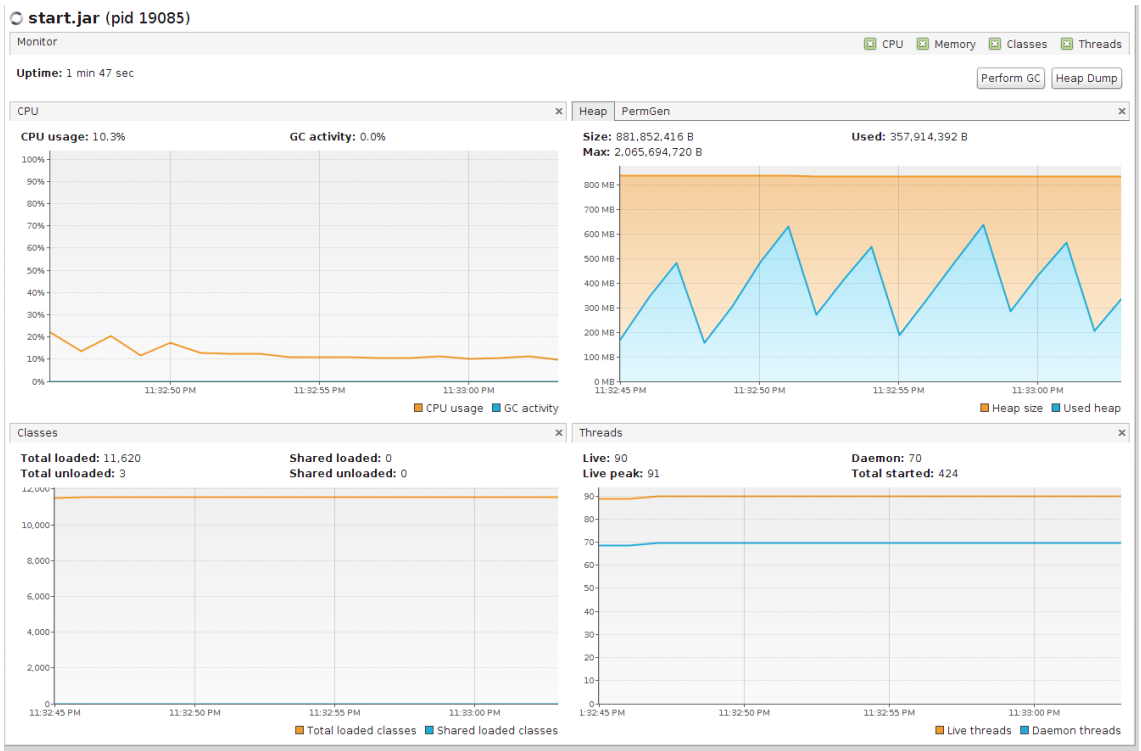

Figure A.97: Resources used when running Exp6 with 240 messages per minute

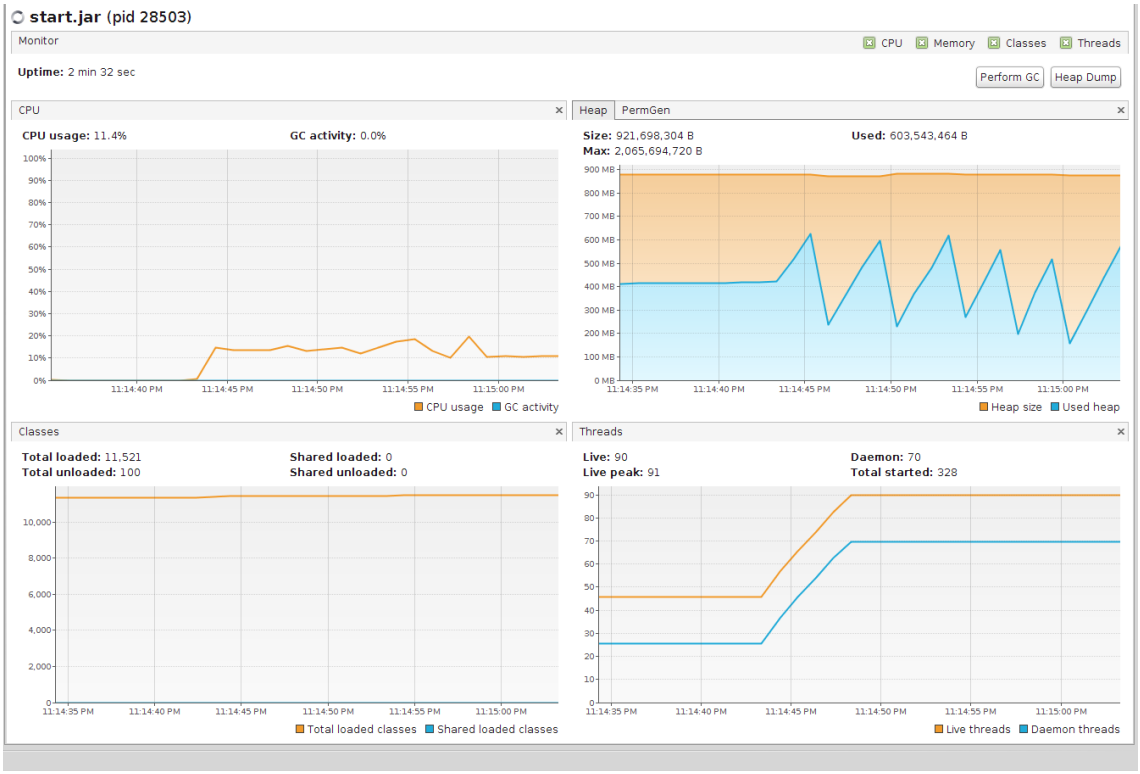

Figure A.98: Resources used when running Exp6 with 480 messages per minute

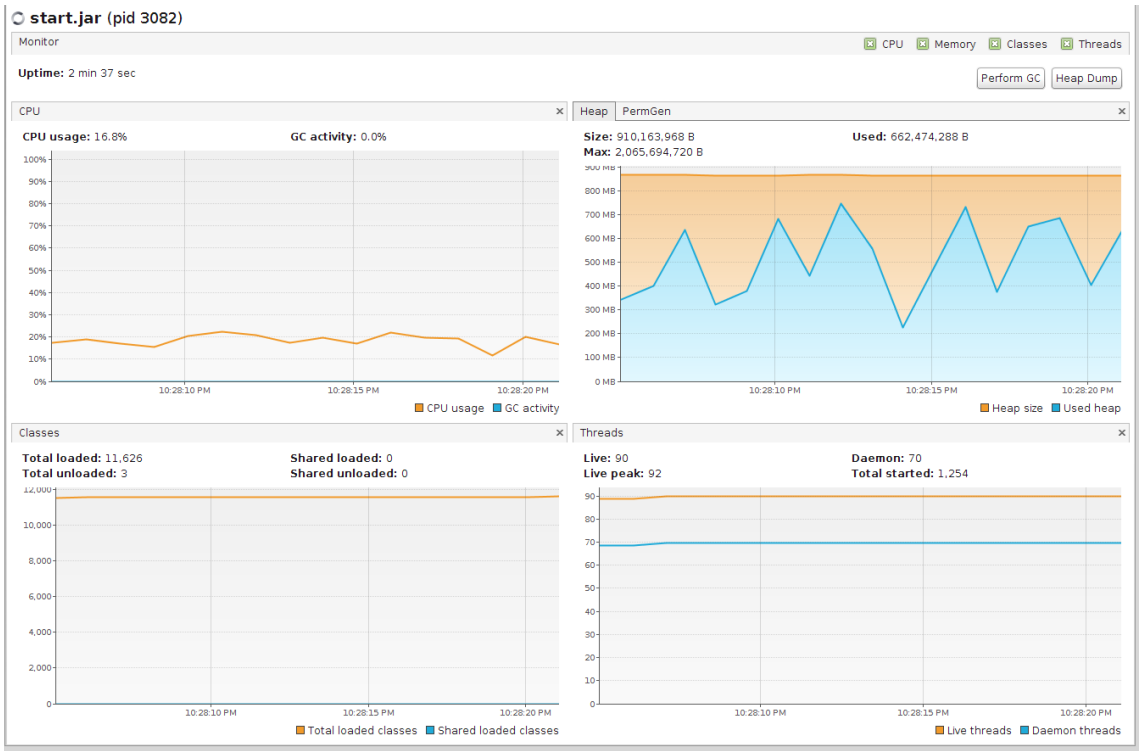

Figure A.99: Resources used when running Exp6 with 1000 messages per minute

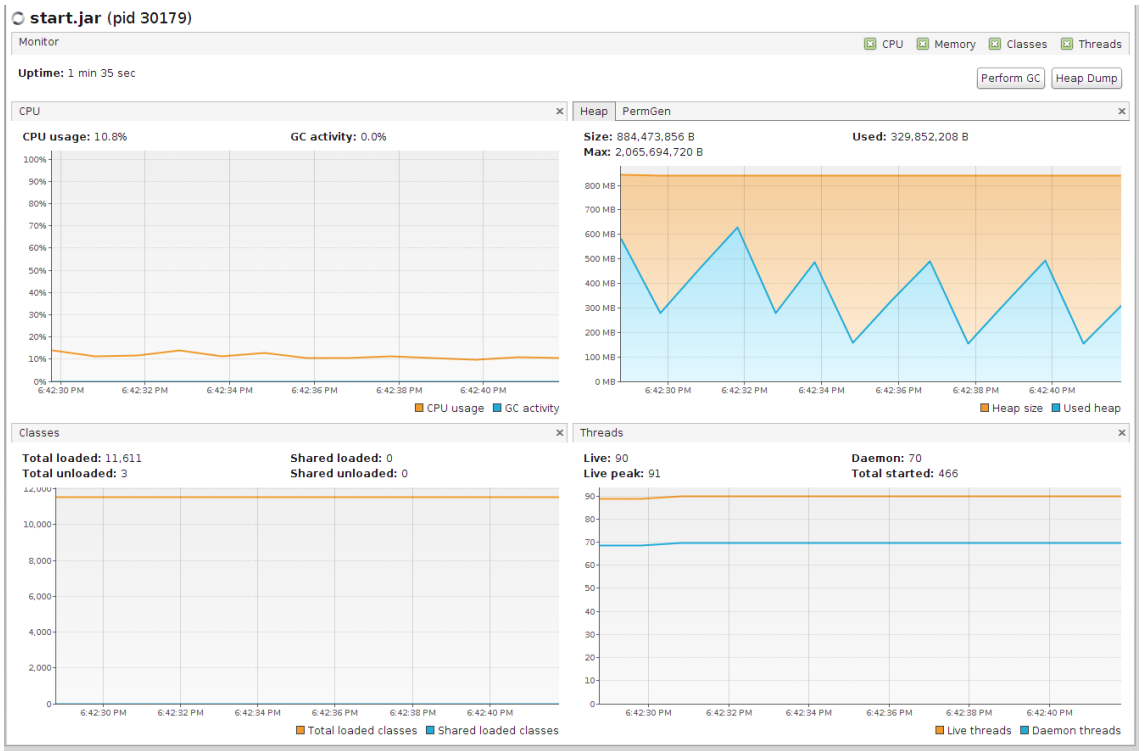

Figure A.100: Resources used when running Exp6 with 2000 messages per minute

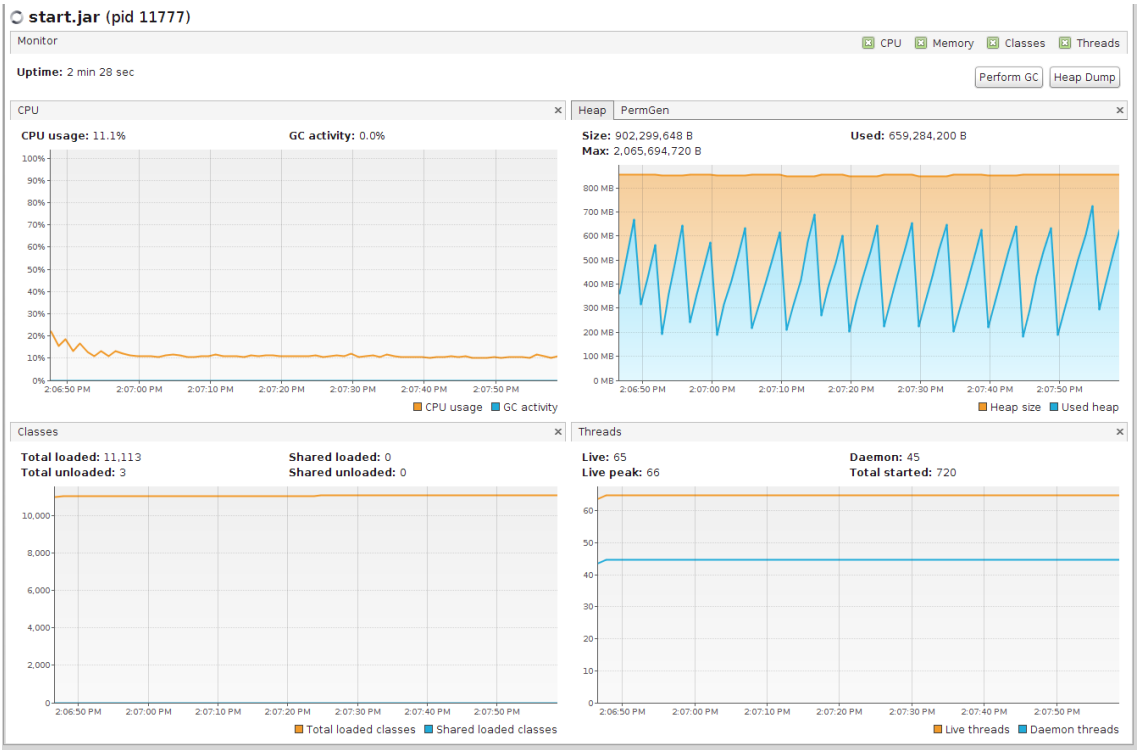

Figure A.101: Resources used when running Exp6 with 4000 messages per minute

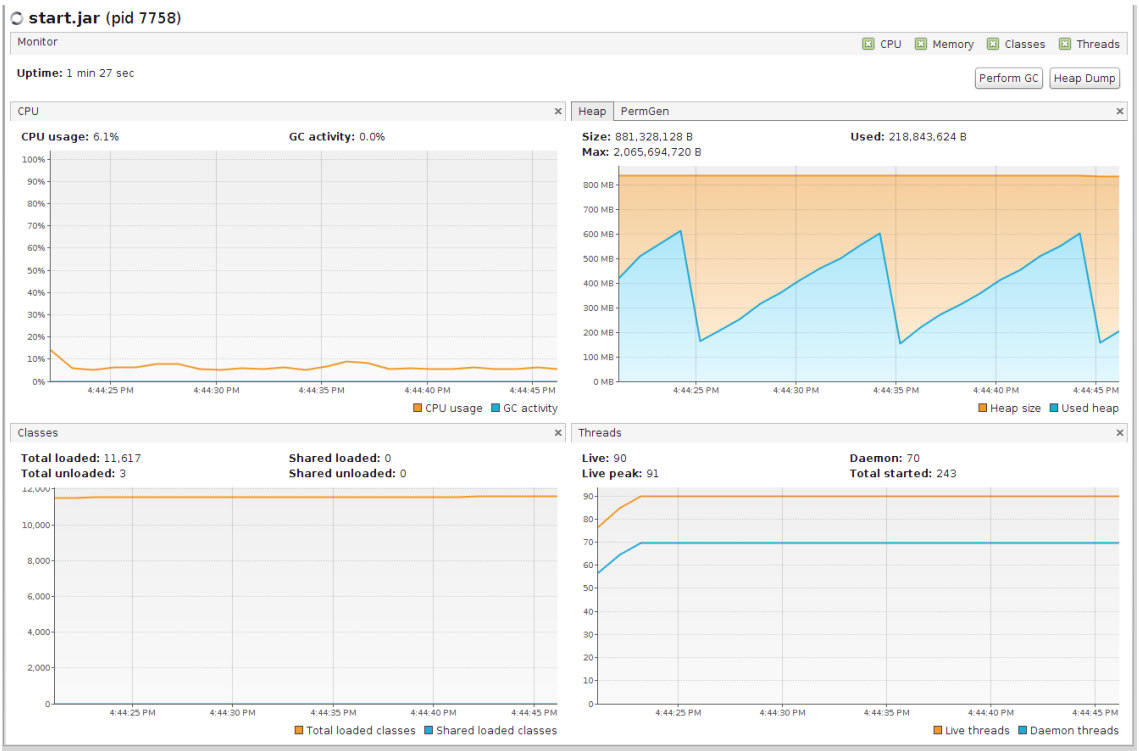

Figure A.102: Resources used when running Exp7 with 60 messages per minute

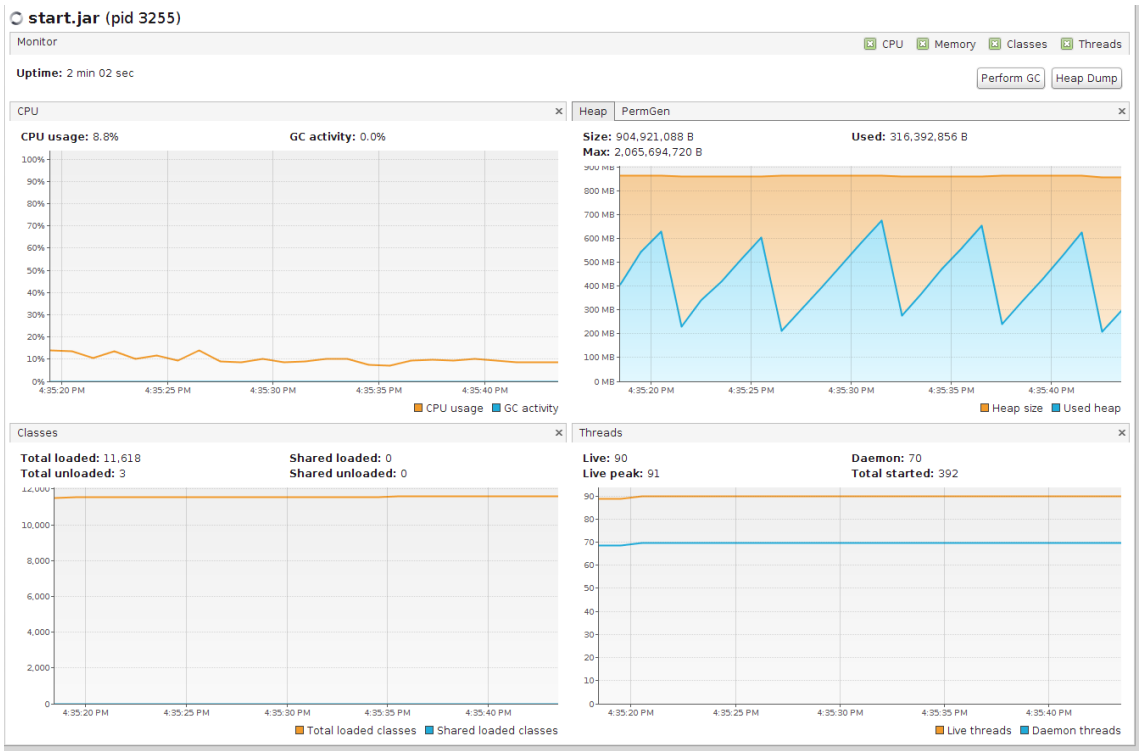

Figure A.103: Resources used when running Exp7 with 120 messages per minute

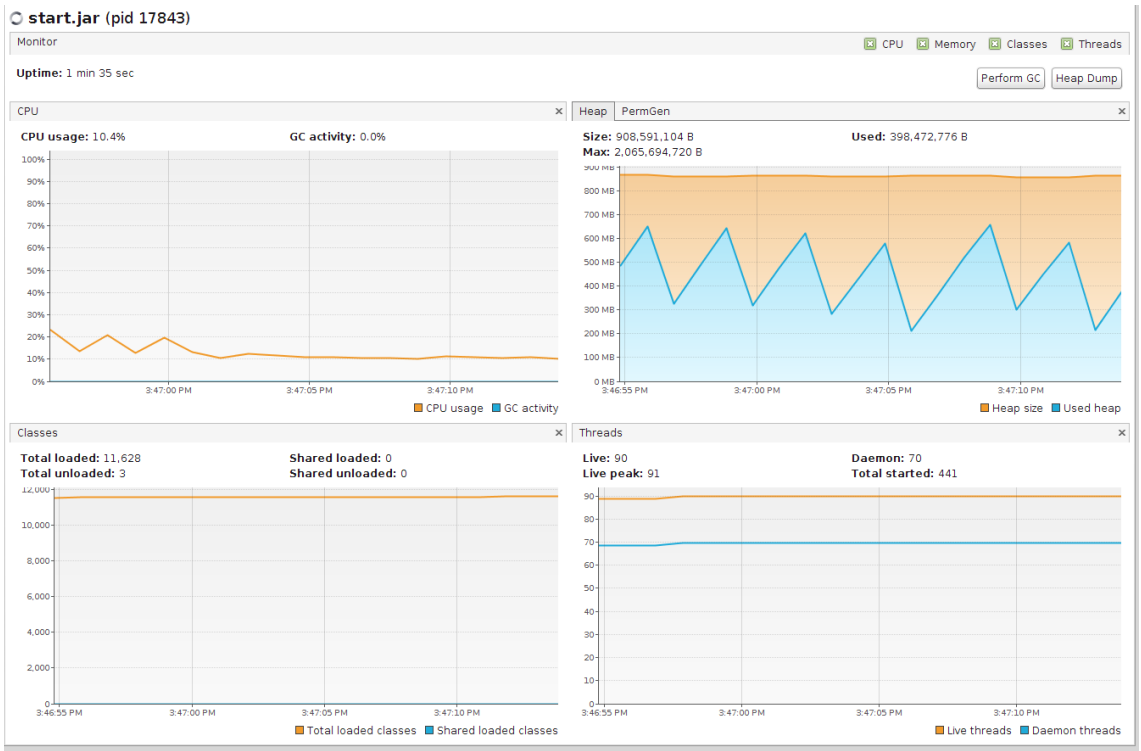

Figure A.104: Resources used when running Exp7 with 240 messages per minute

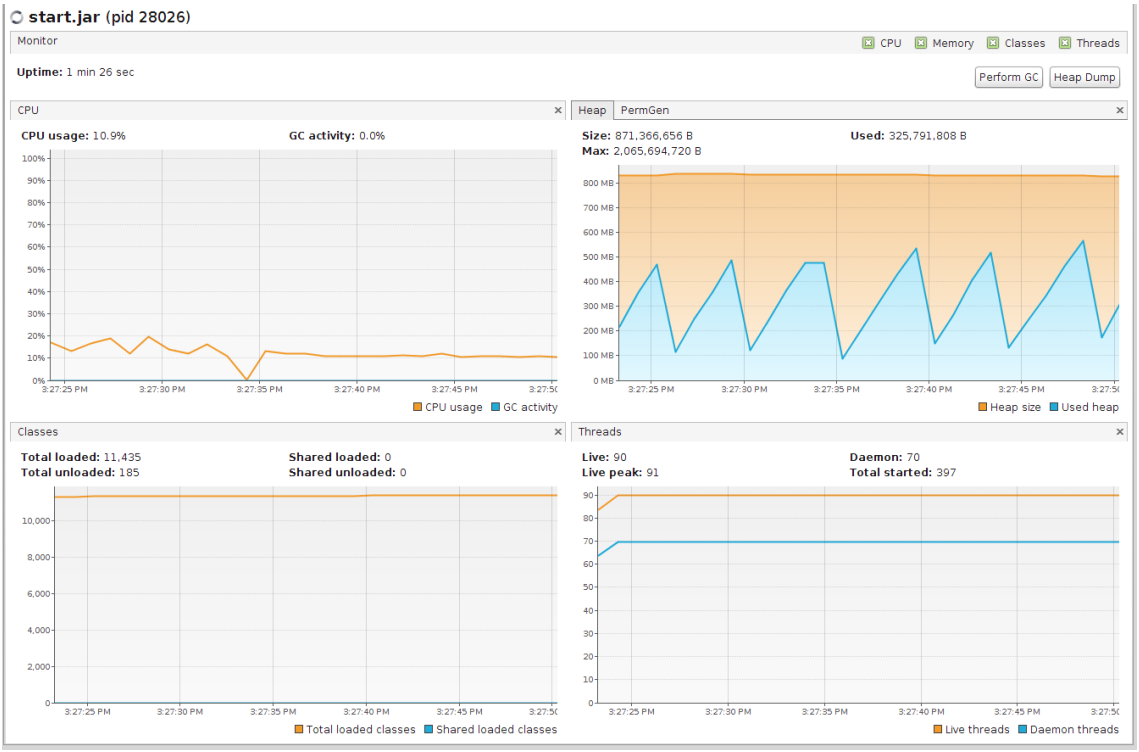

Figure A.105: Resources used when running Exp7 with 480 messages per minute

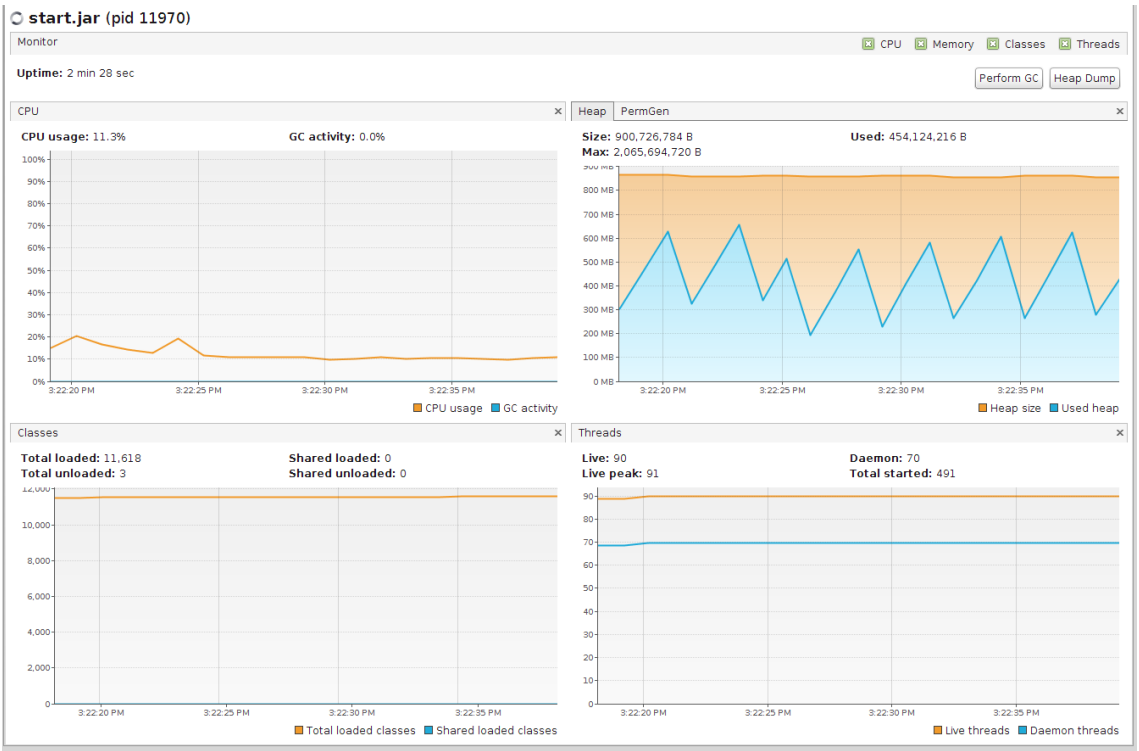

Figure A.106: Resources used when running Exp7 with 1000 messages per minute

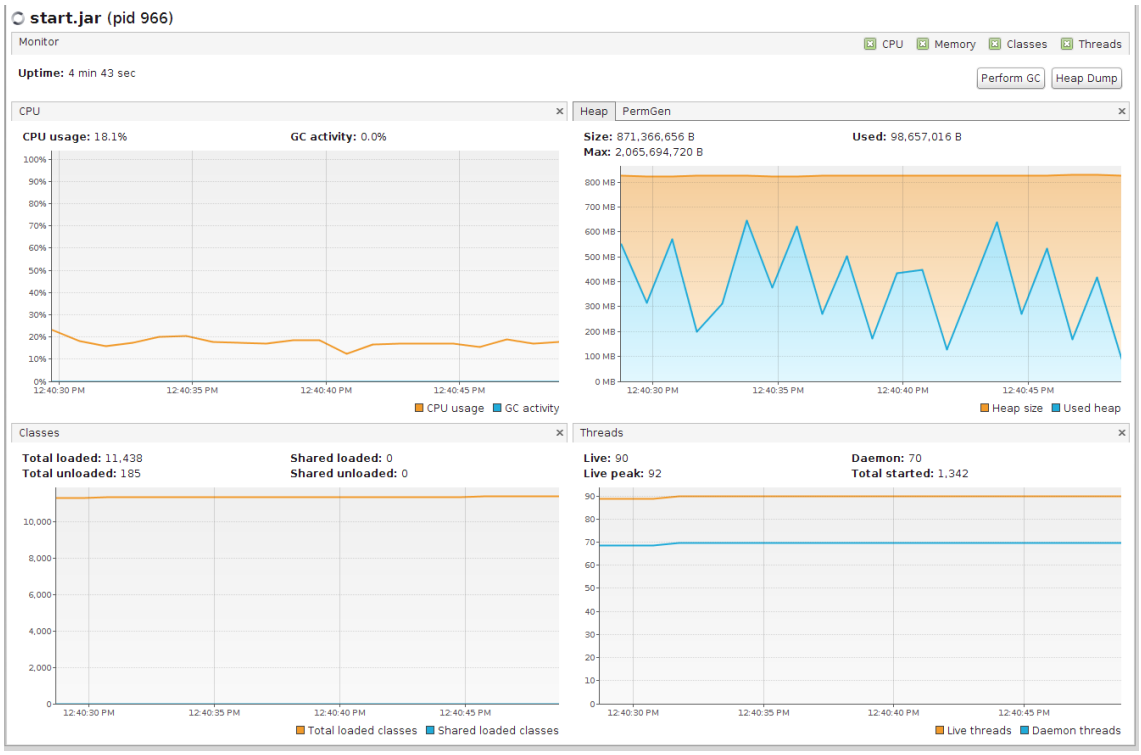

Figure A.107: Resources used when running Exp7 with 2000 messages per minute

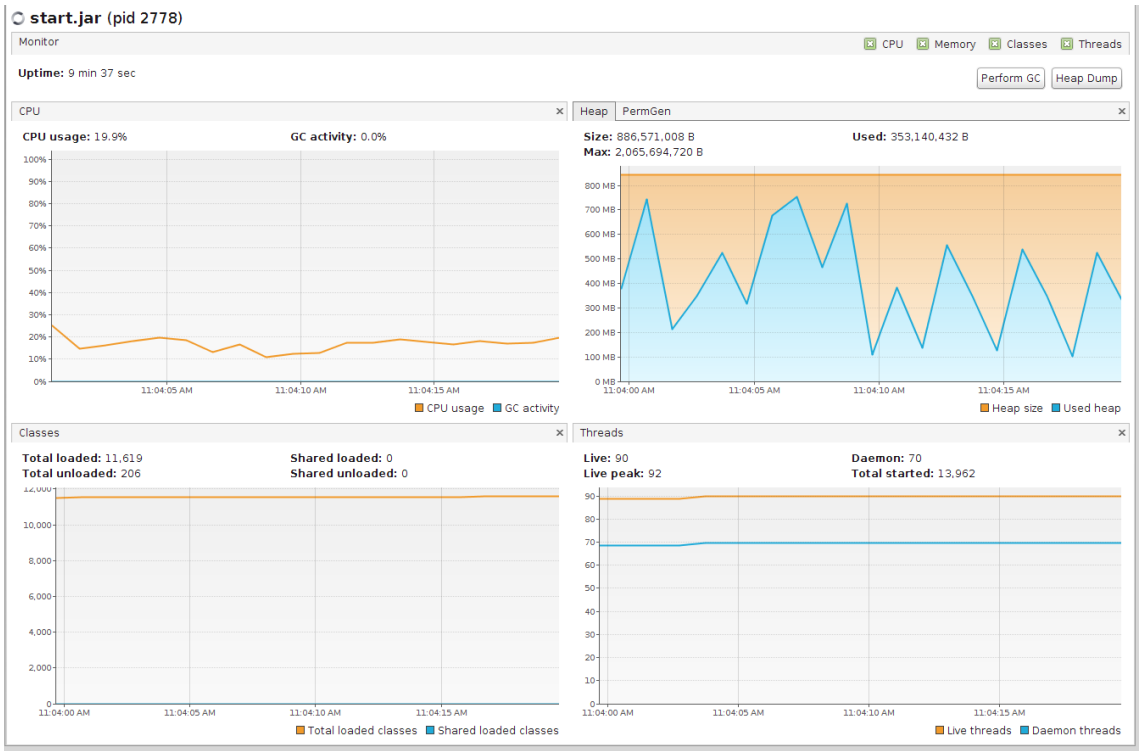

Figure A.108: Resources used when running Exp7 with 4000 messages per minute

A.3.2 Comparision graphs of middleware delay

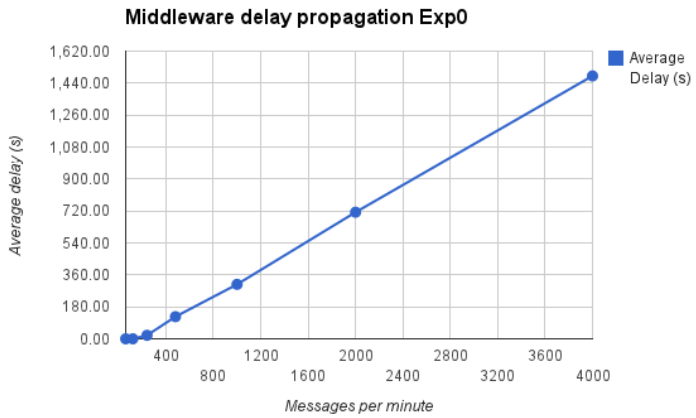

Figure A.109: Middleware delay propagation using Exp0

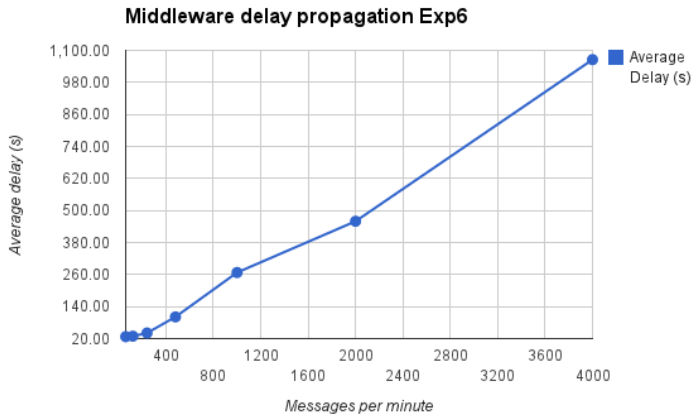

Figure A.110: Middleware delay propagation using Exp6

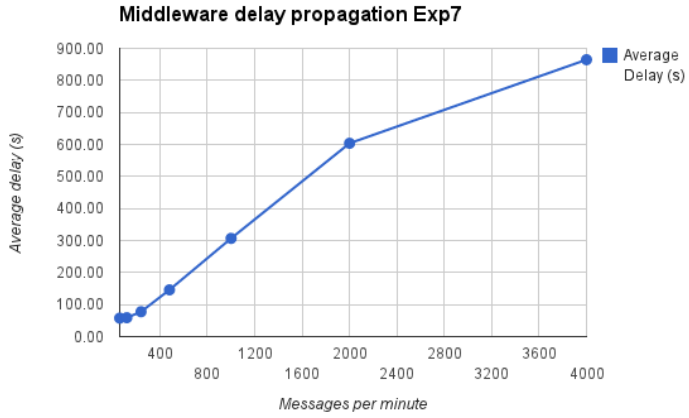

Figure A.111: Middleware delay propagation using Exp7

A.4 1 source, 1 session, 1 client, SEDA full info

A.4.1 visualVM monitor screenshots

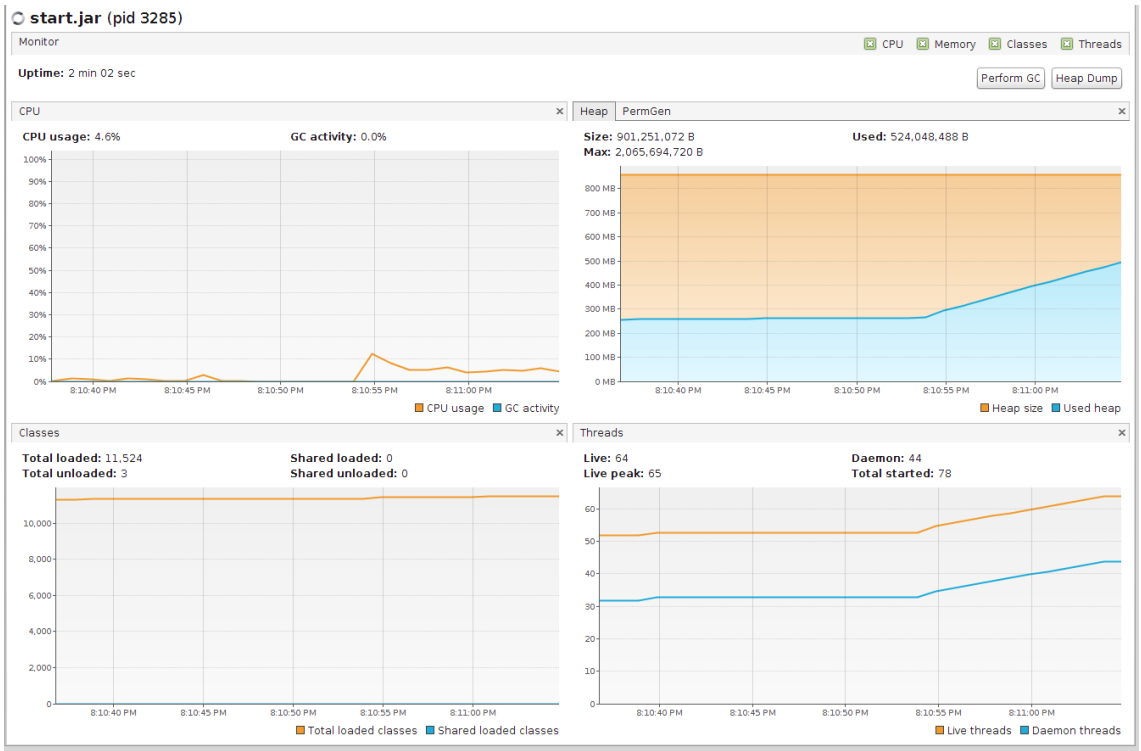

Figure A.112: Resources used when running Exp0 with 60 messages per minute

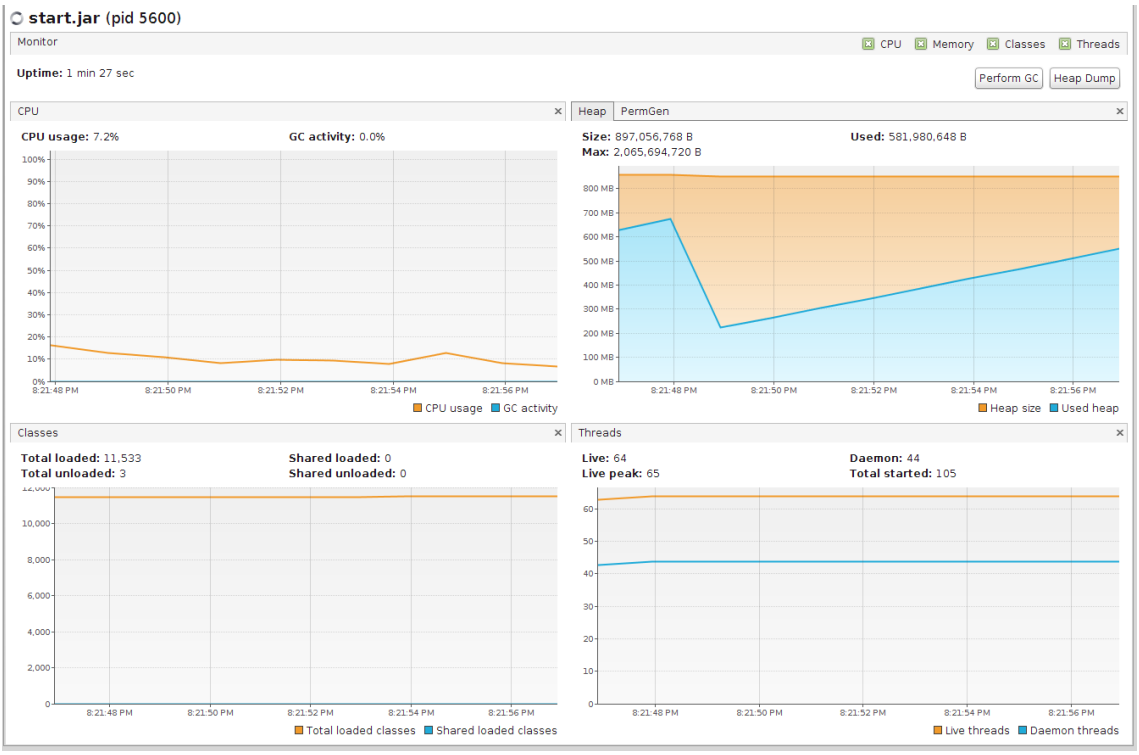

Figure A.113: Resources used when running Exp0 with 120 messages per minute

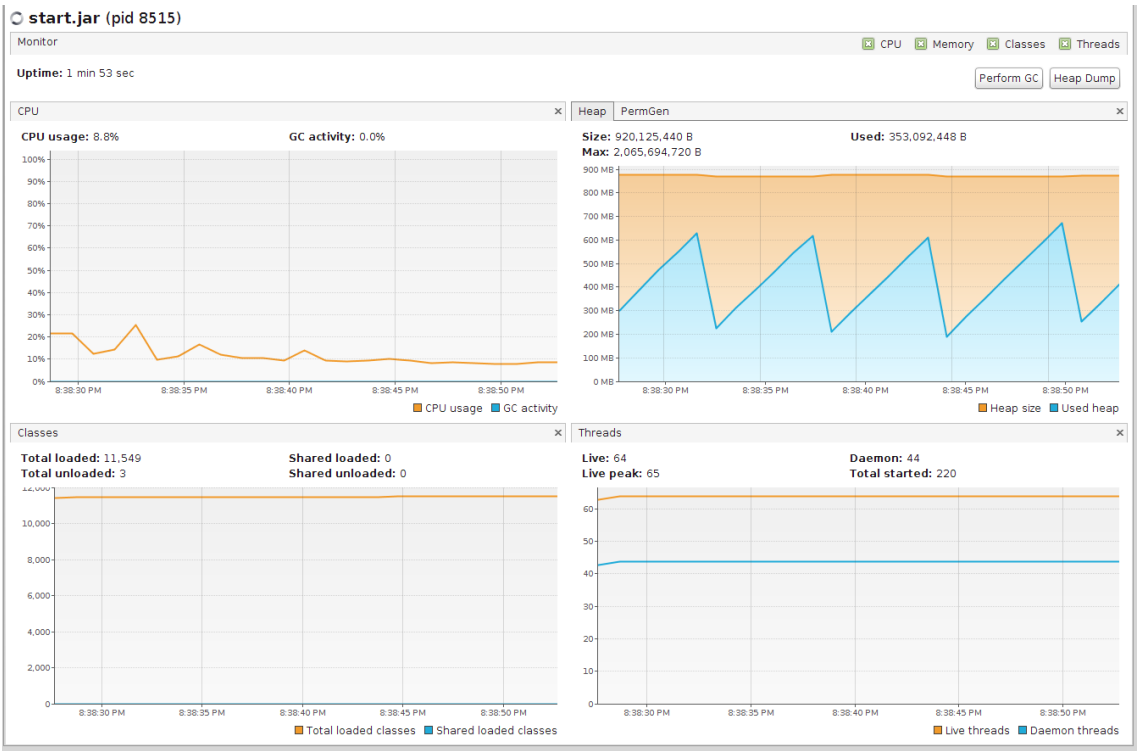

Figure A.114: Resources used when running Exp0 with 240 messages per minute

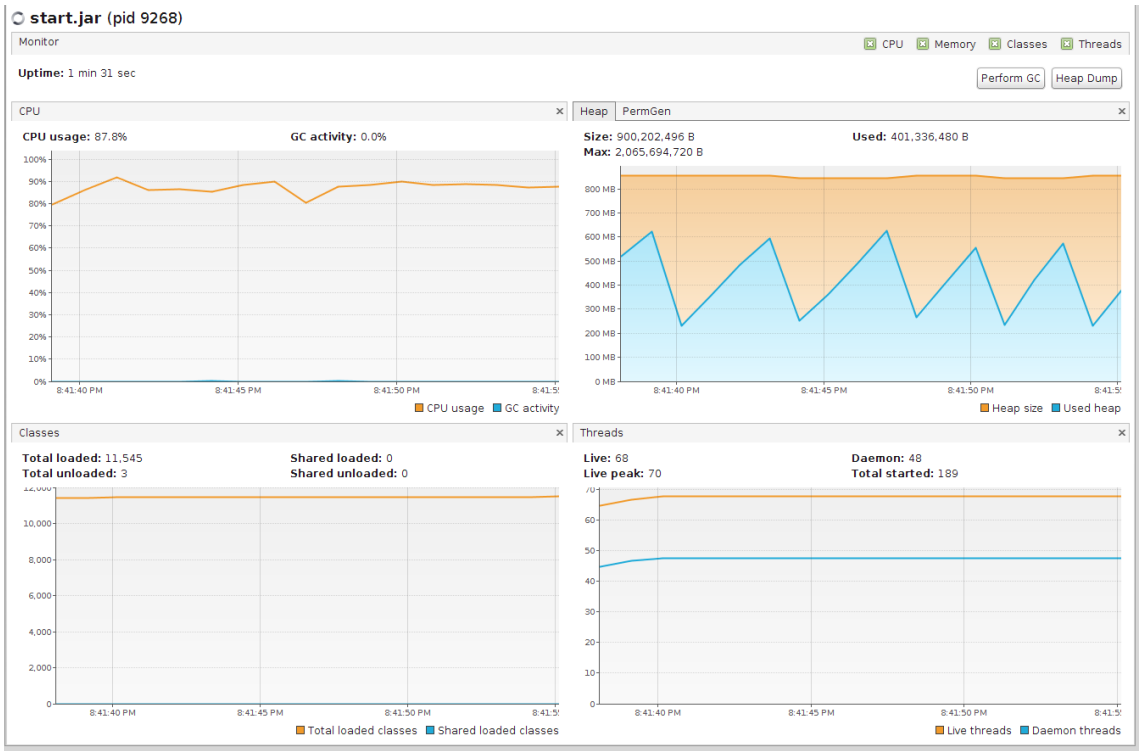

Figure A.115: Resources used when running Exp0 with 480 messages per minute

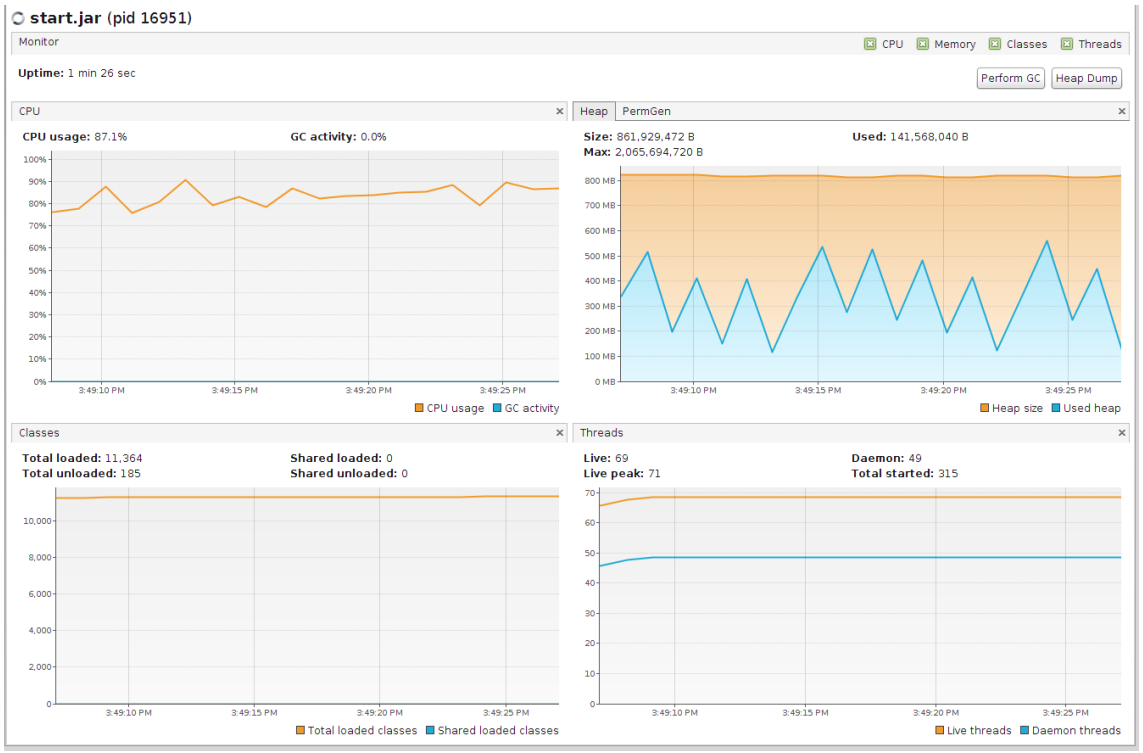

Figure A.116: Resources used when running Exp0 with 1000 messages per minute

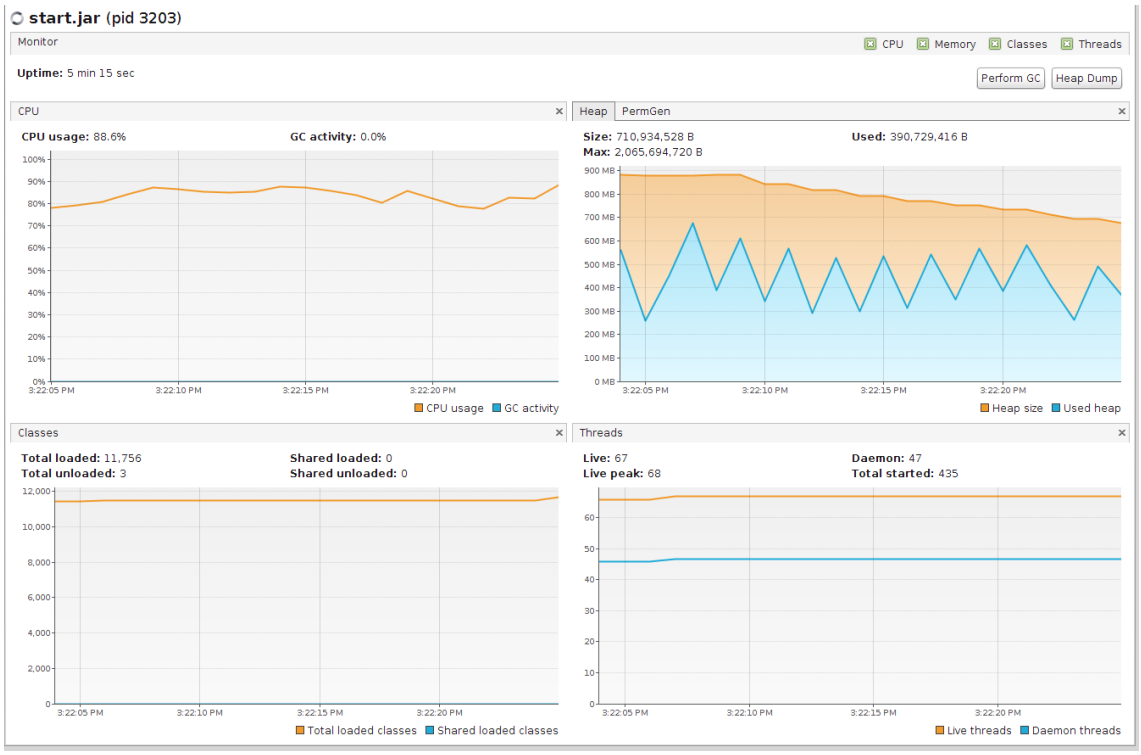

Figure A.117: Resources used when running Exp0 with 2000 messages per minute

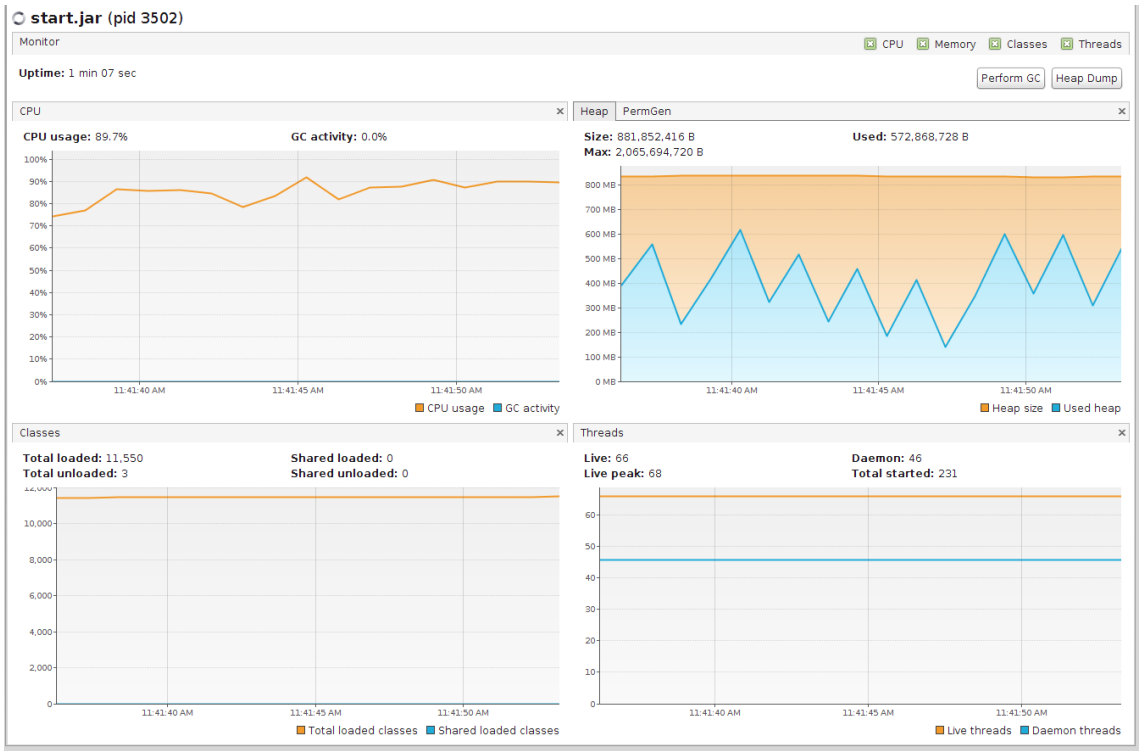

Figure A.118: Resources used when running Exp0 with 4000 messages per minute

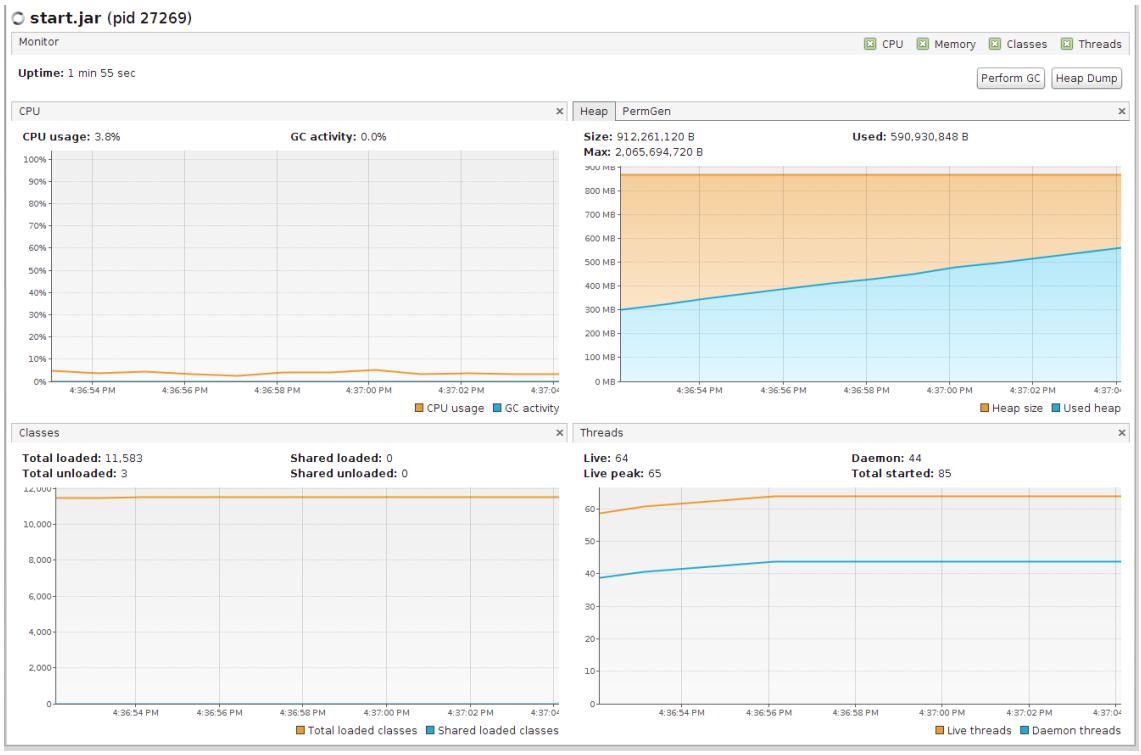

Figure A.119: Resources used when running Exp6 with 60 messages per minute

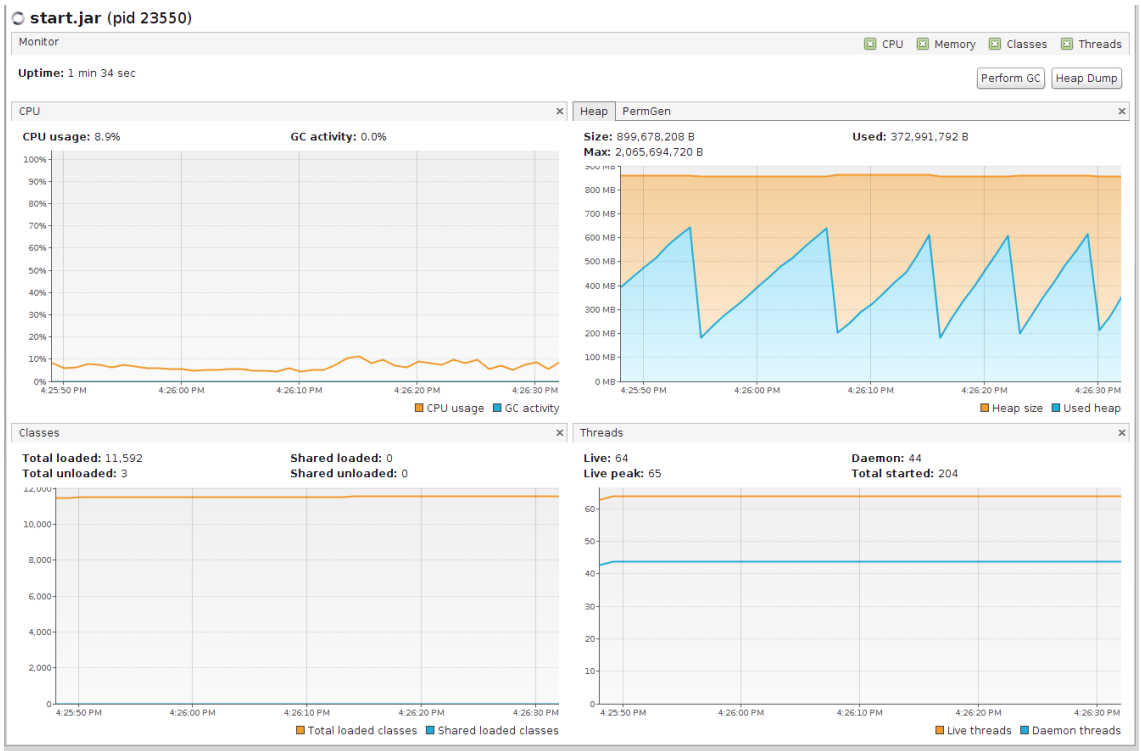

Figure A.120: Resources used when running Exp6 with 120 messages per minute

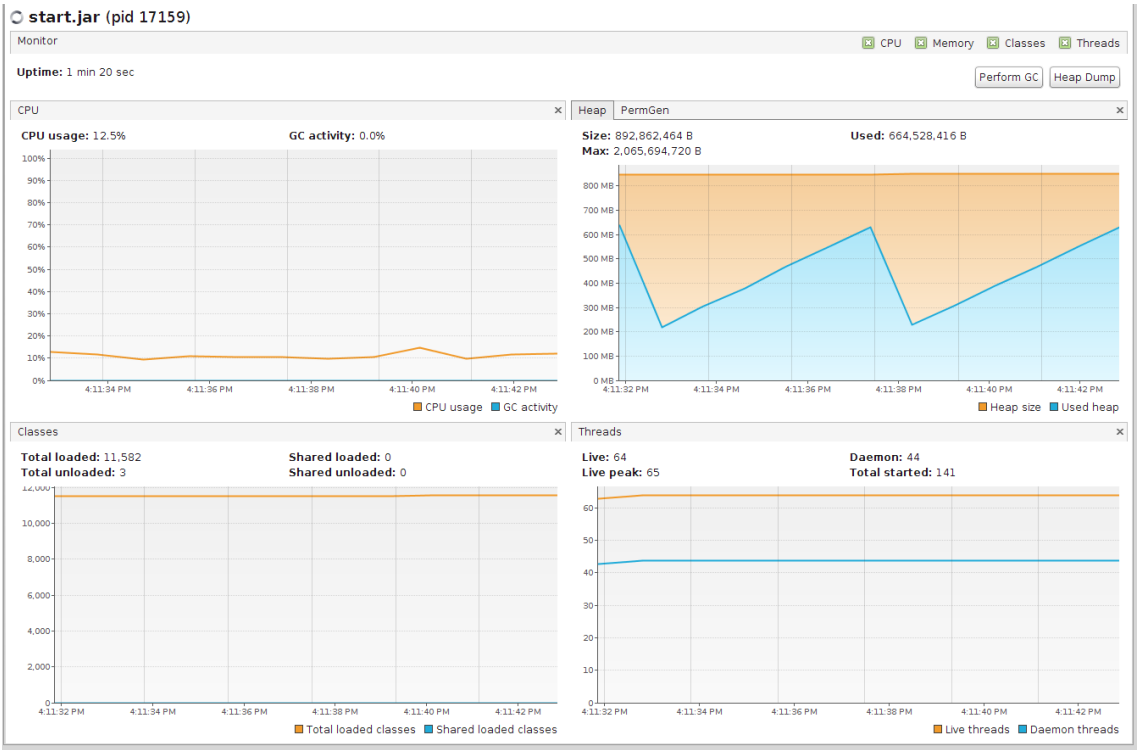

Figure A.121: Resources used when running Exp6 with 240 messages per minute

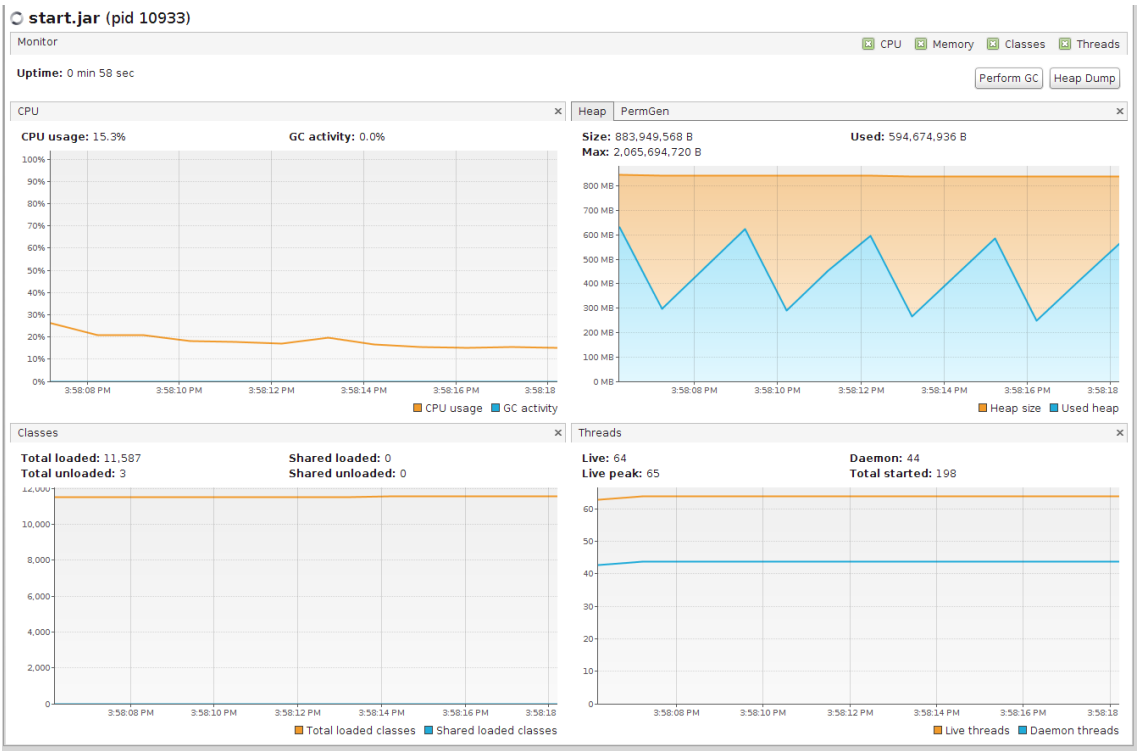

Figure A.122: Resources used when running Exp6 with 480 messages per minute

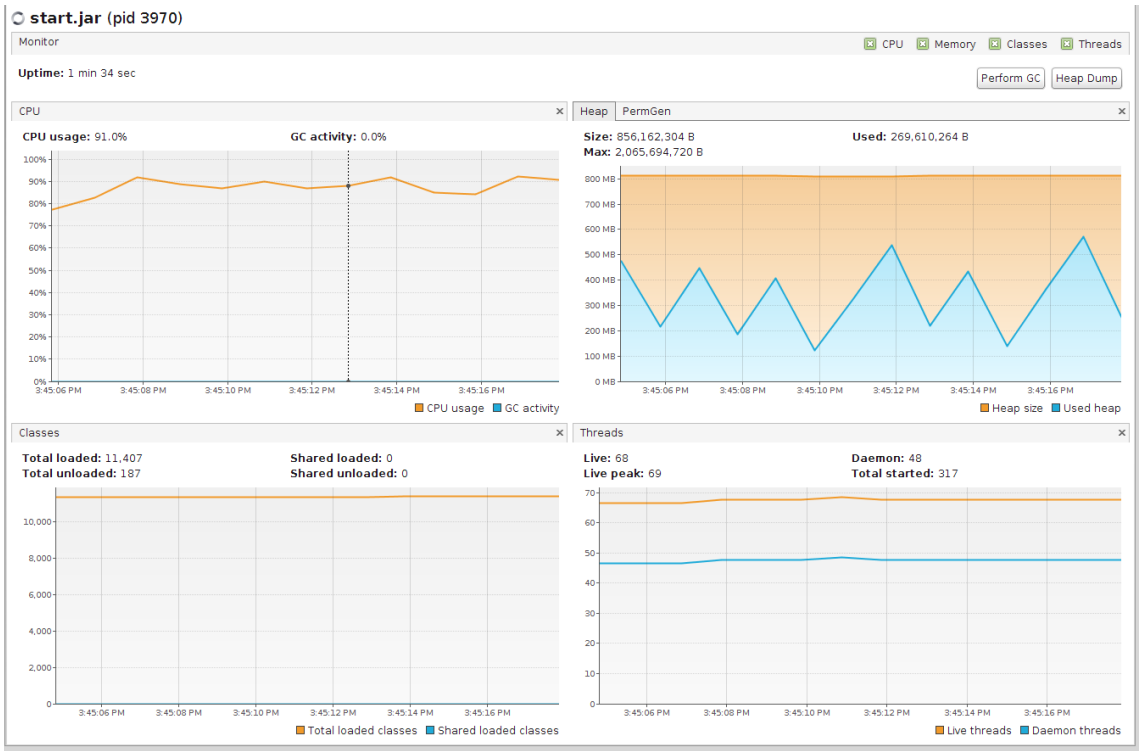

Figure A.123: Resources used when running Exp6 with 1000 messages per minute

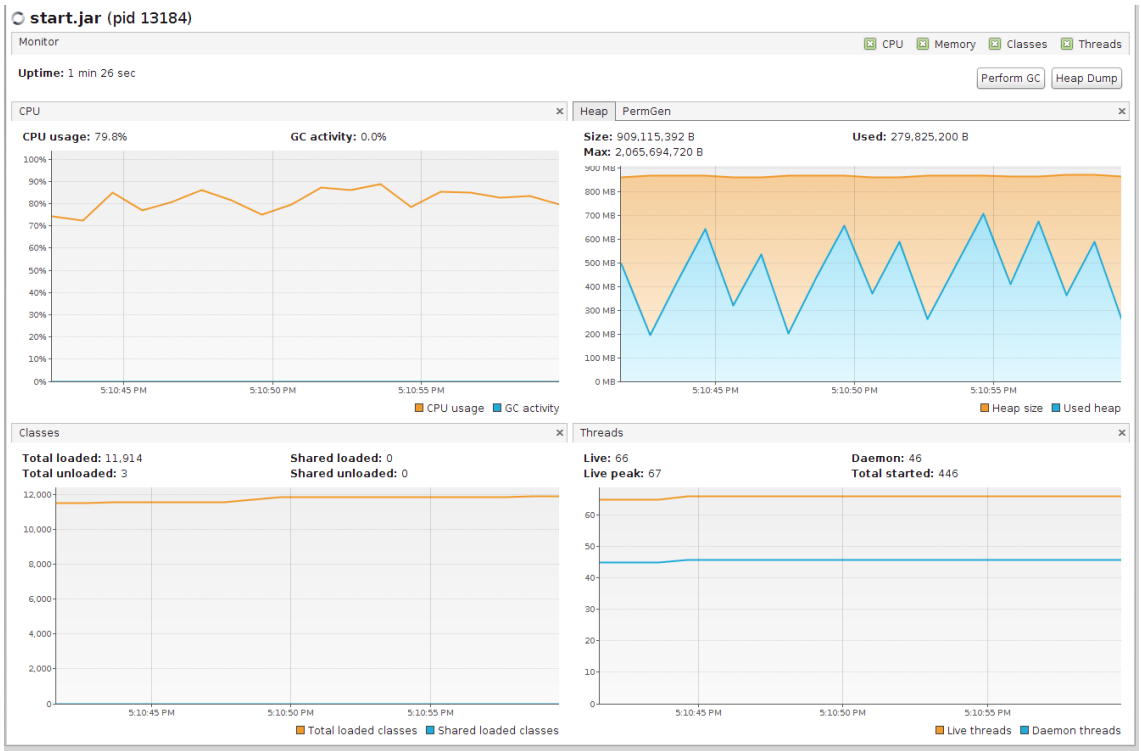

Figure A.124: Resources used when running Exp6 with 2000 messages per minute

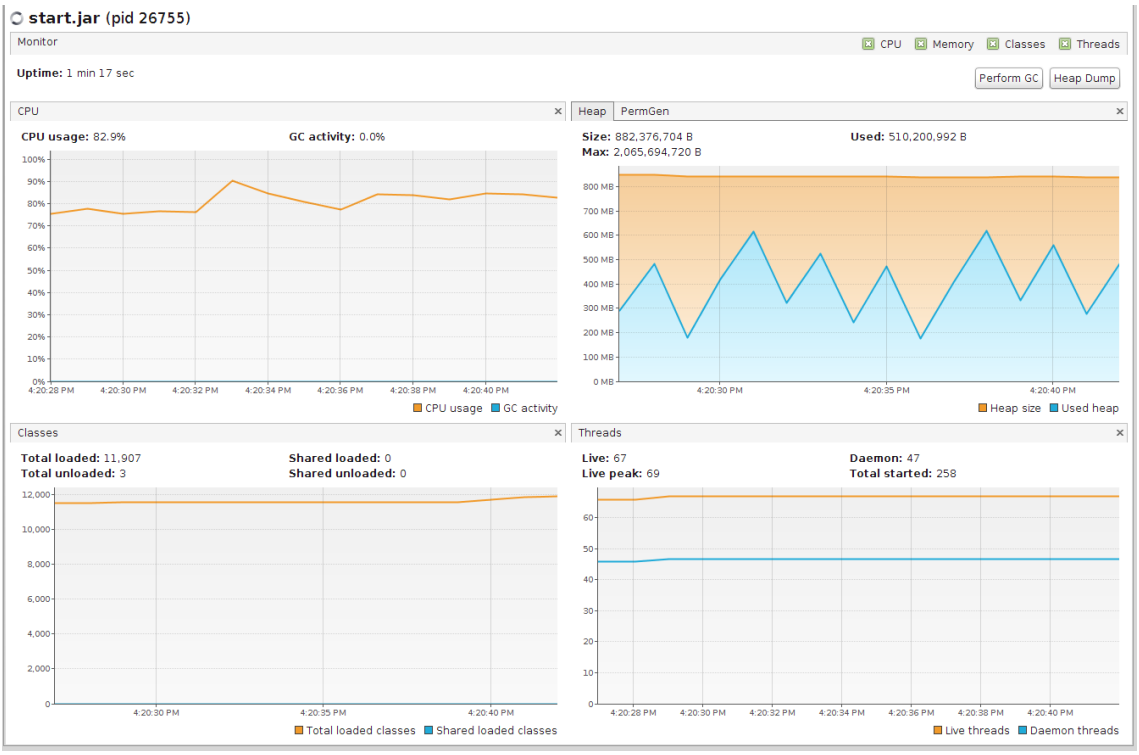

Figure A.125: Resources used when running Exp6 with 4000 messages per minute

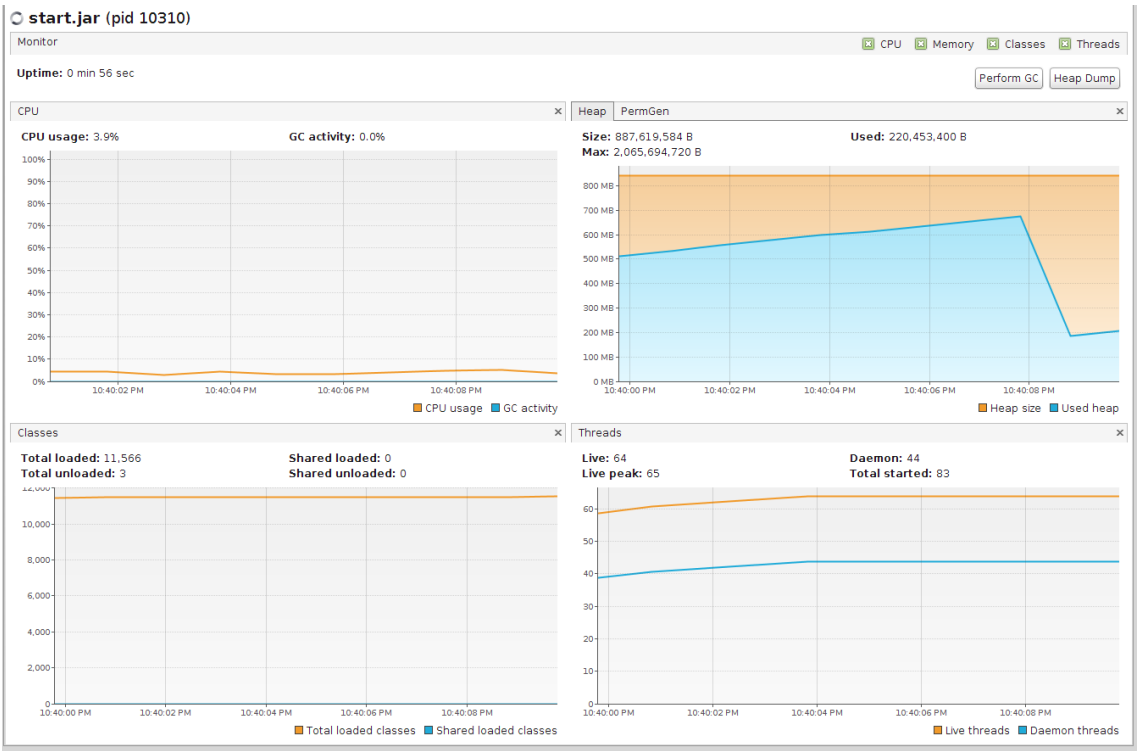

Figure A.126: Resources used when running Exp7 with 60 messages per minute

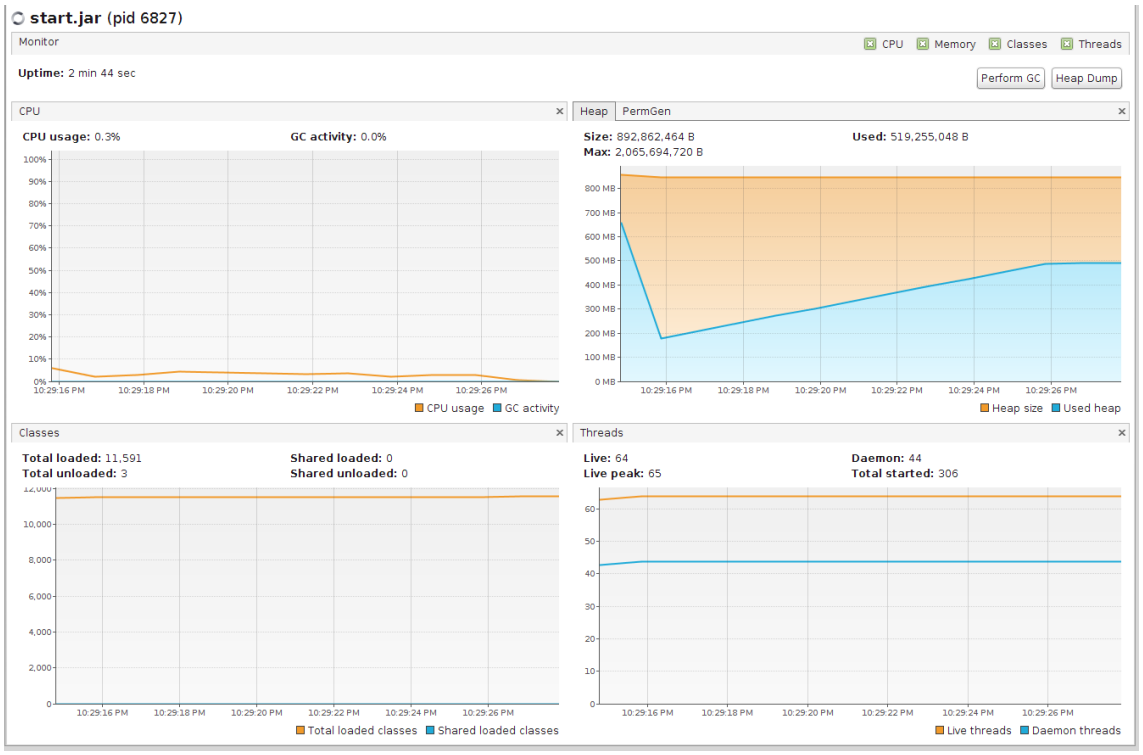

Figure A.127: Resources used when running Exp7 with 120 messages per minute

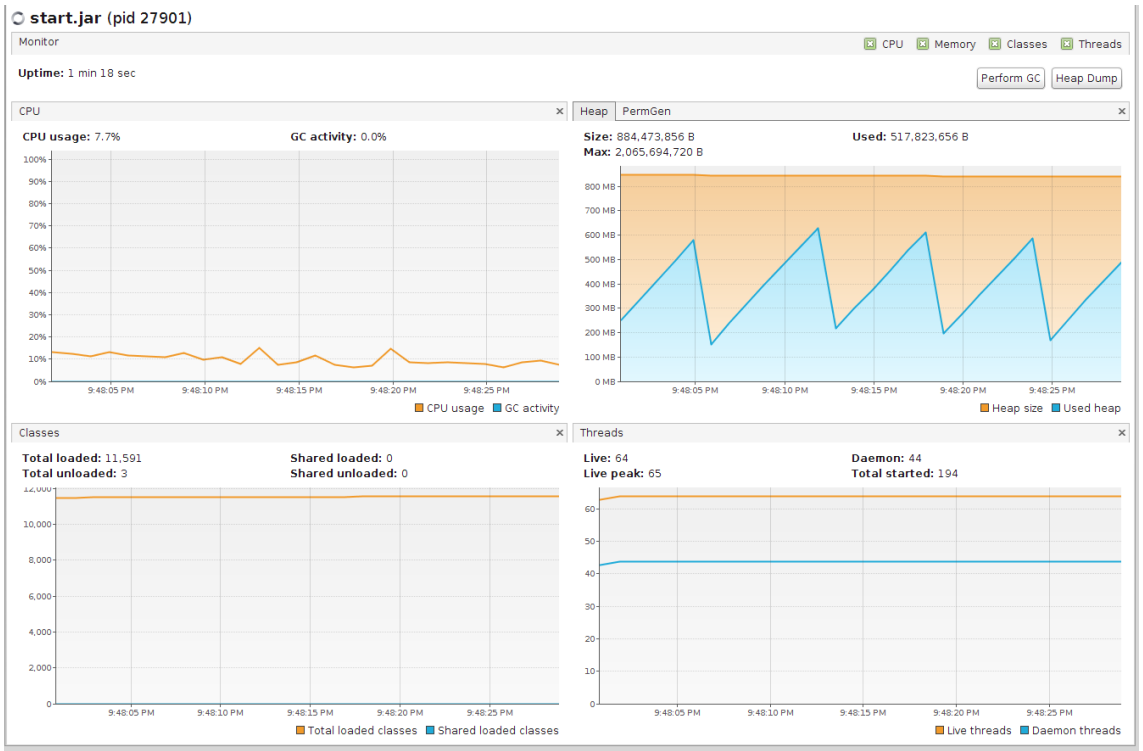

Figure A.128: Resources used when running Exp7 with 240 messages per minute

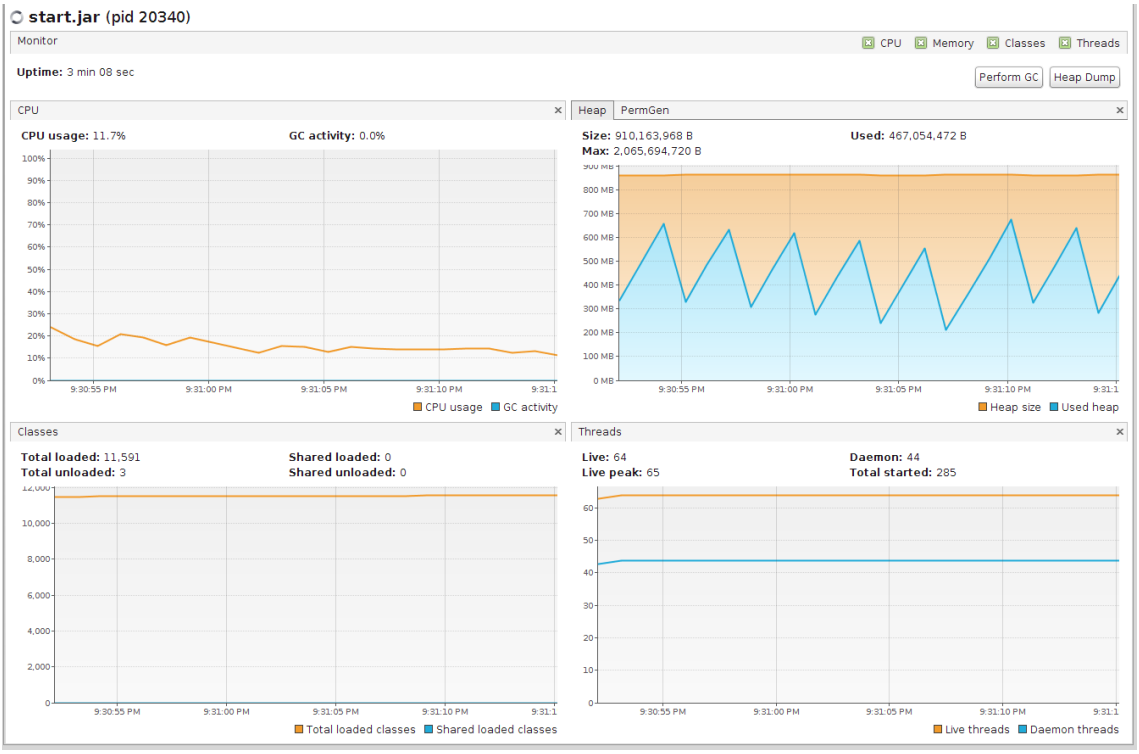

Figure A.129: Resources used when running Exp7 with 480 messages per minute

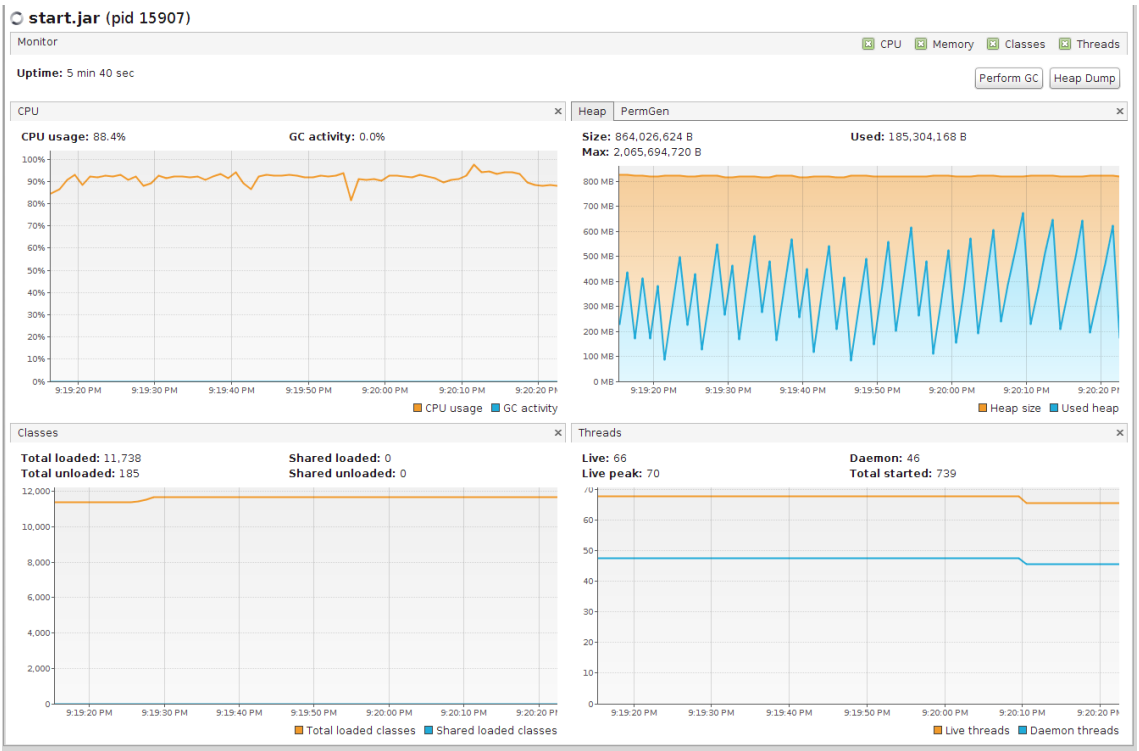

Figure A.130: Resources used when running Exp7 with 1000 messages per minute

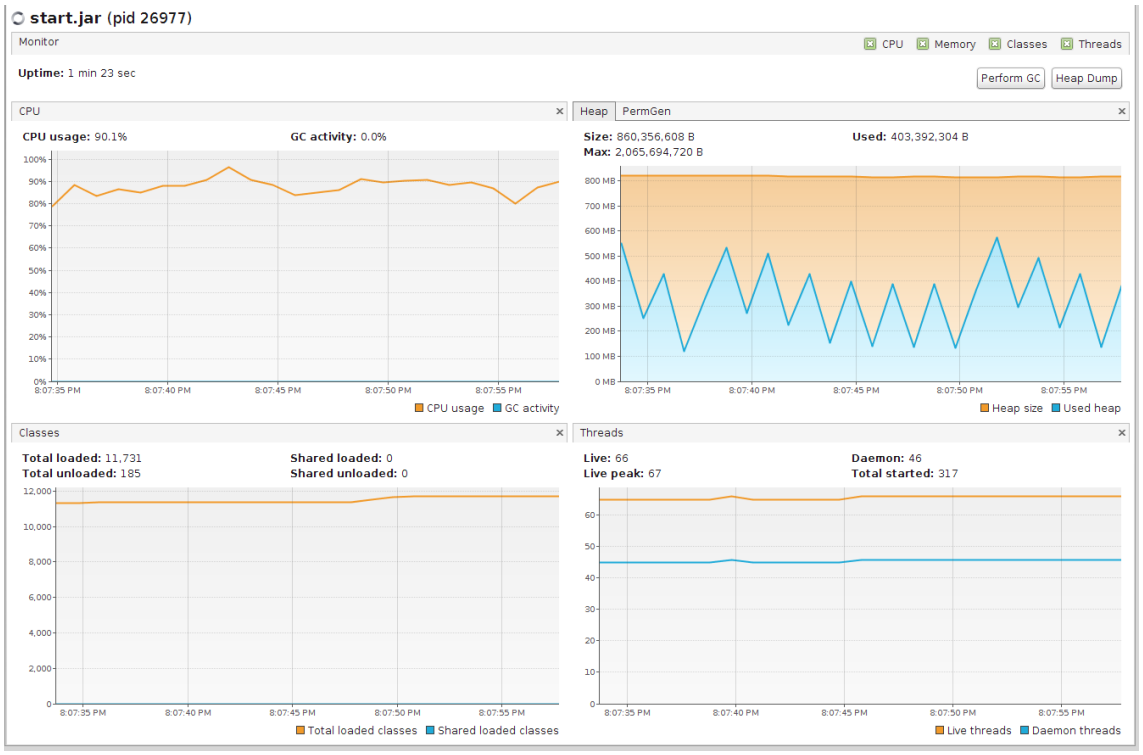

Figure A.131: Resources used when running Exp7 with 2000 messages per minute

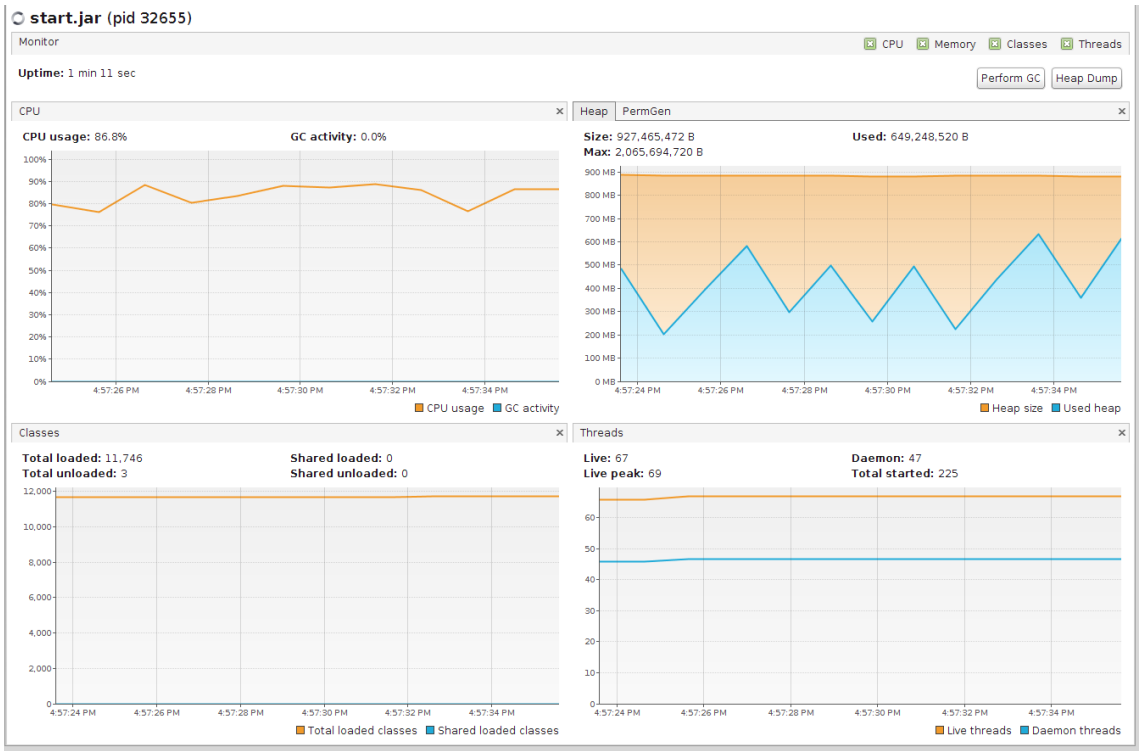

Figure A.132: Resources used when running Exp7 with 4000 messages per minute

A.4.2 Comparision graphs of middleware delay

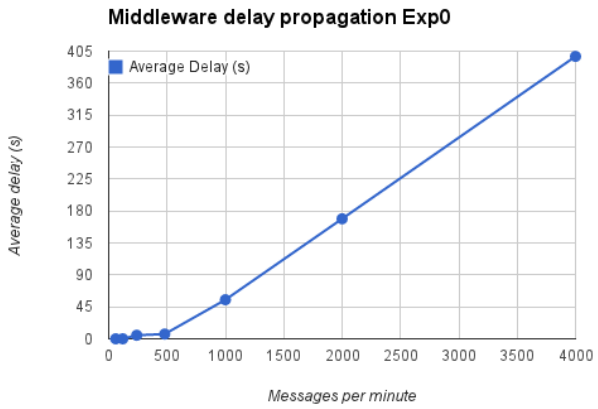

Figure A.133: Middleware delay propagation using Exp0

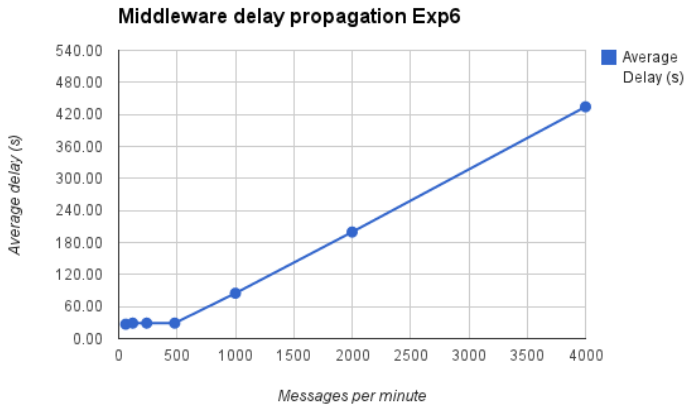

Figure A.134: Middleware delay propagation using Exp6

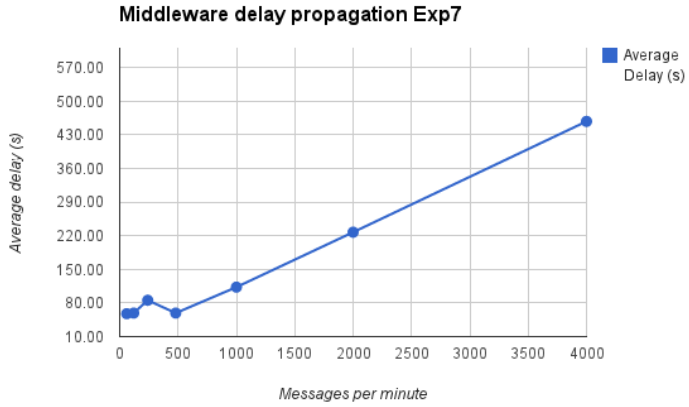

Figure A.135: Middleware delay propagation using Exp7

A.5 1 source, 2 sessions, 1 client, SEDA full info

A.5.1 visualVM monitor screenshots

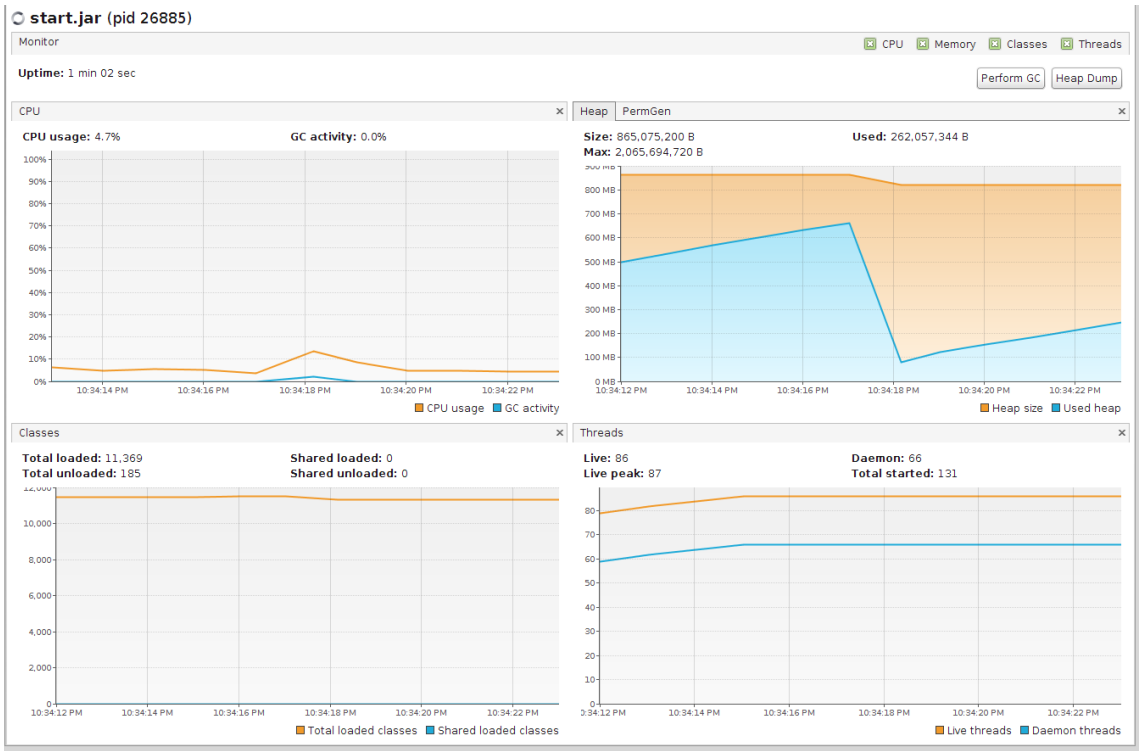

Figure A.136: Resources used when running Exp0 with 60 messages per minute

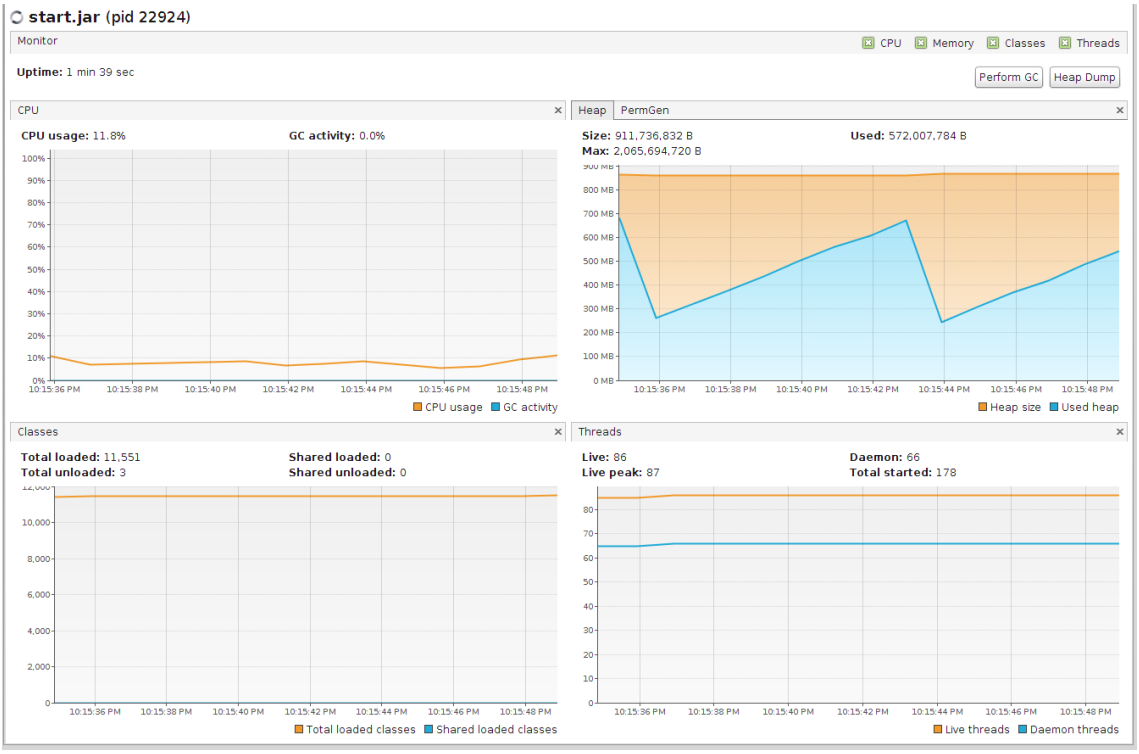

Figure A.137: Resources used when running Exp0 with 120 messages per minute

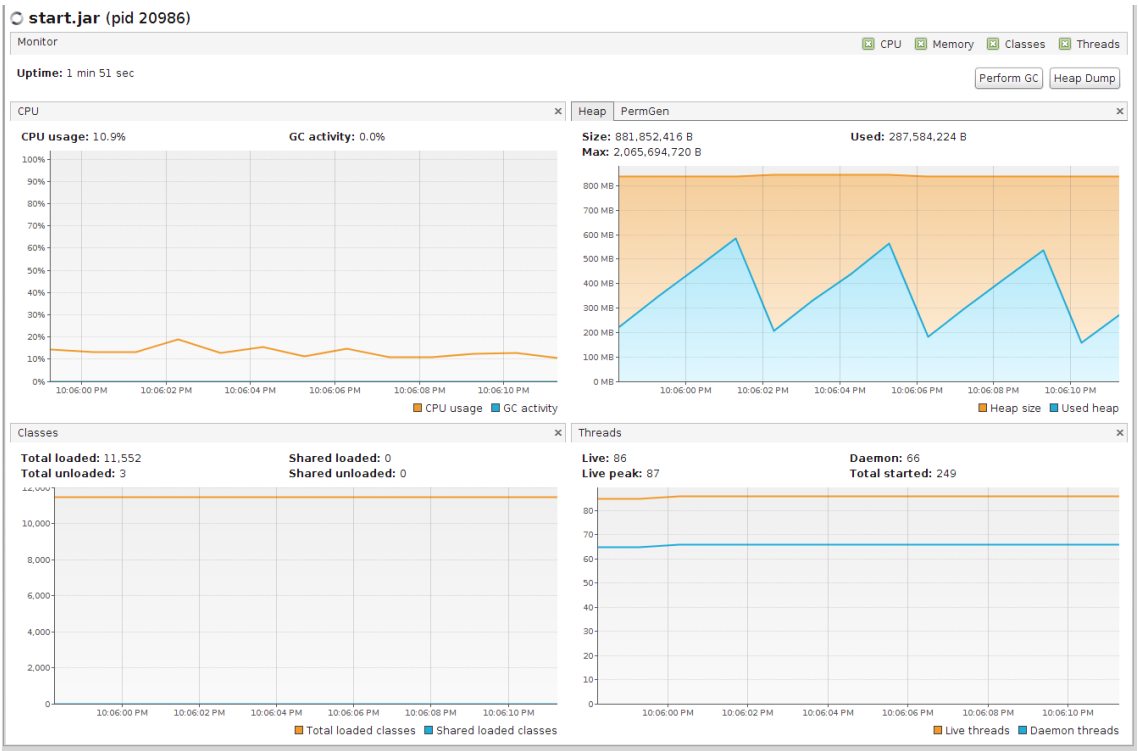

Figure A.138: Resources used when running Exp0 with 240 messages per minute

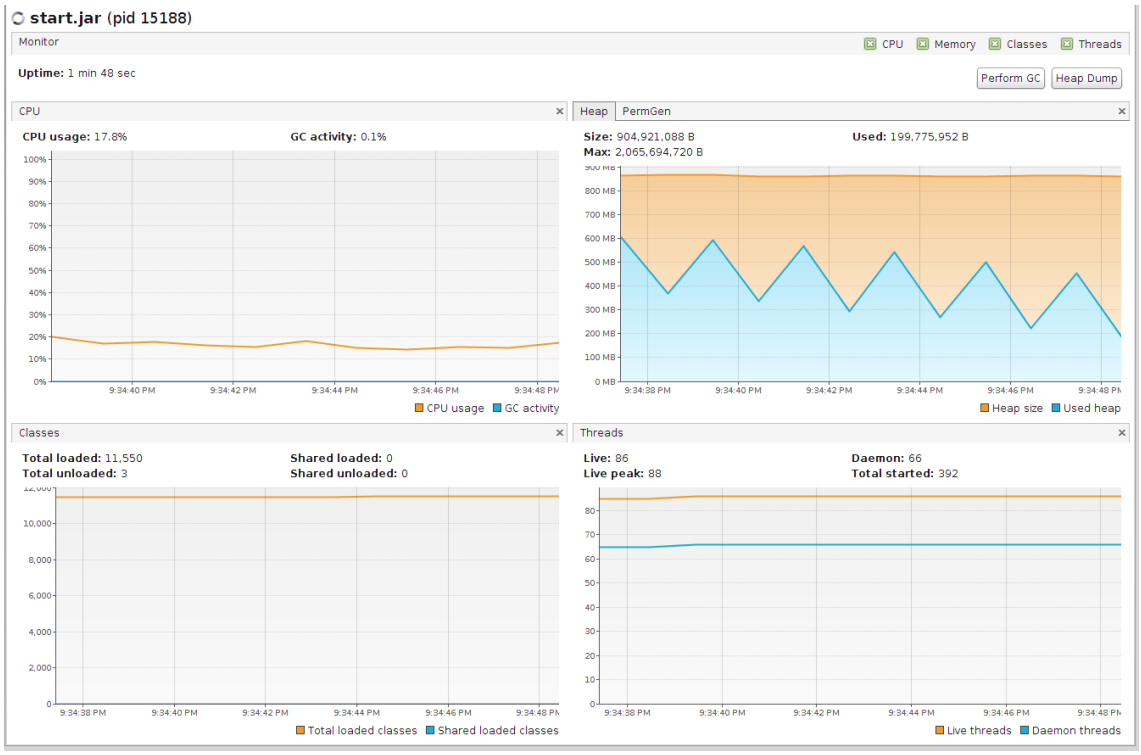

Figure A.139: Resources used when running Exp0 with 480 messages per minute

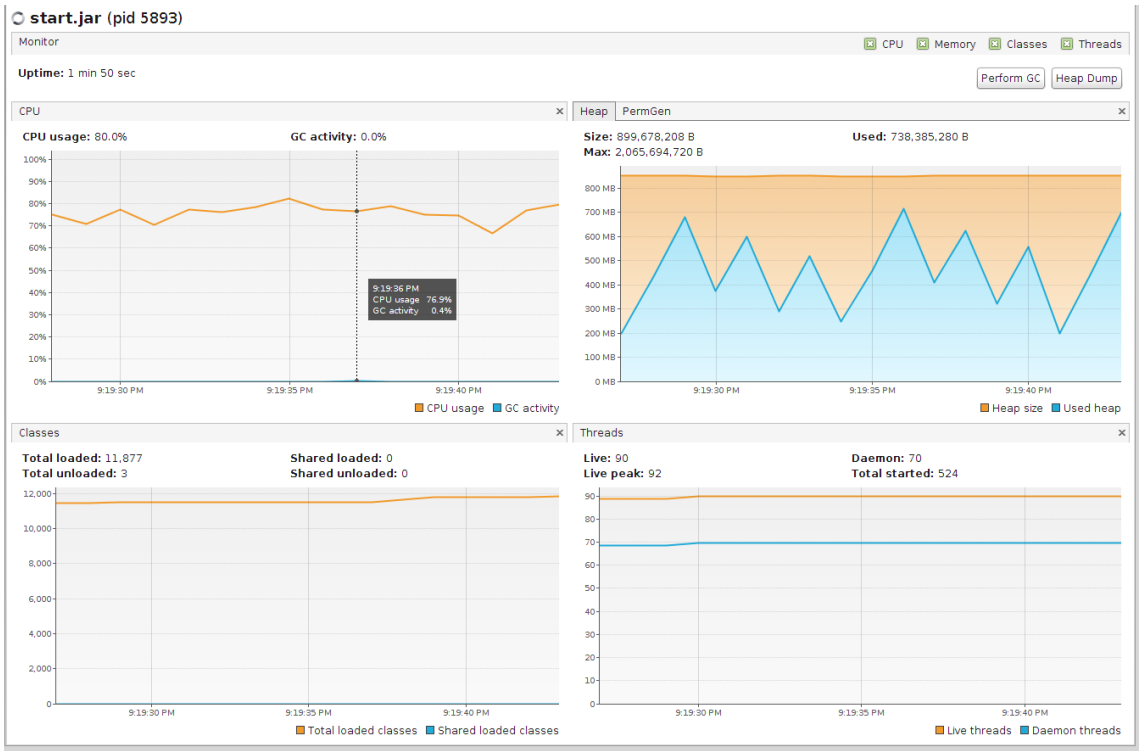

Figure A.140: Resources used when running Exp0 with 1000 messages per minute

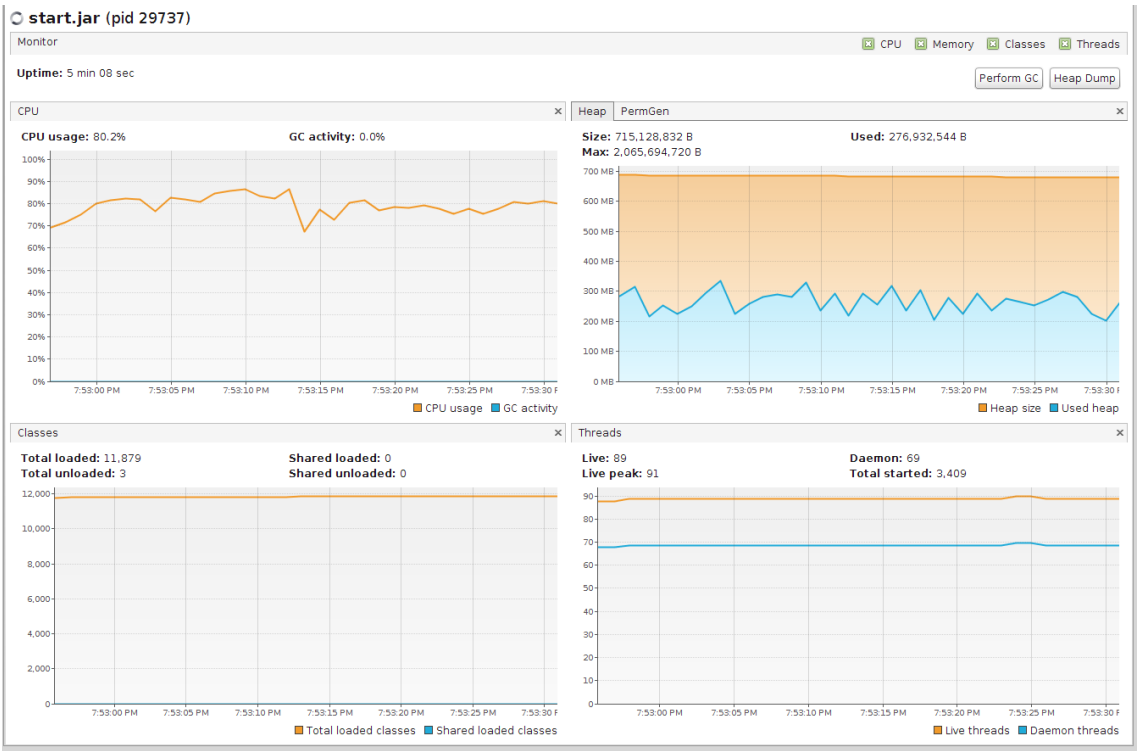

Figure A.141: Resources used when running Exp0 with 2000 messages per minute

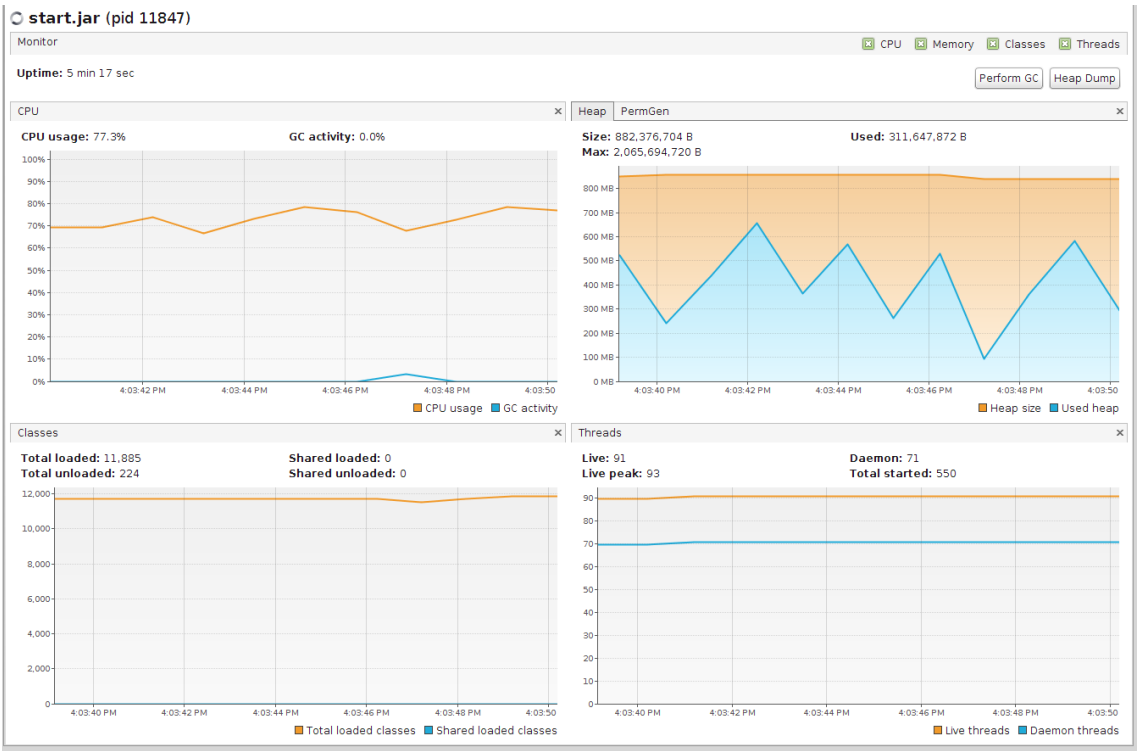

Figure A.142: Resources used when running Exp0 with 4000 messages per minute

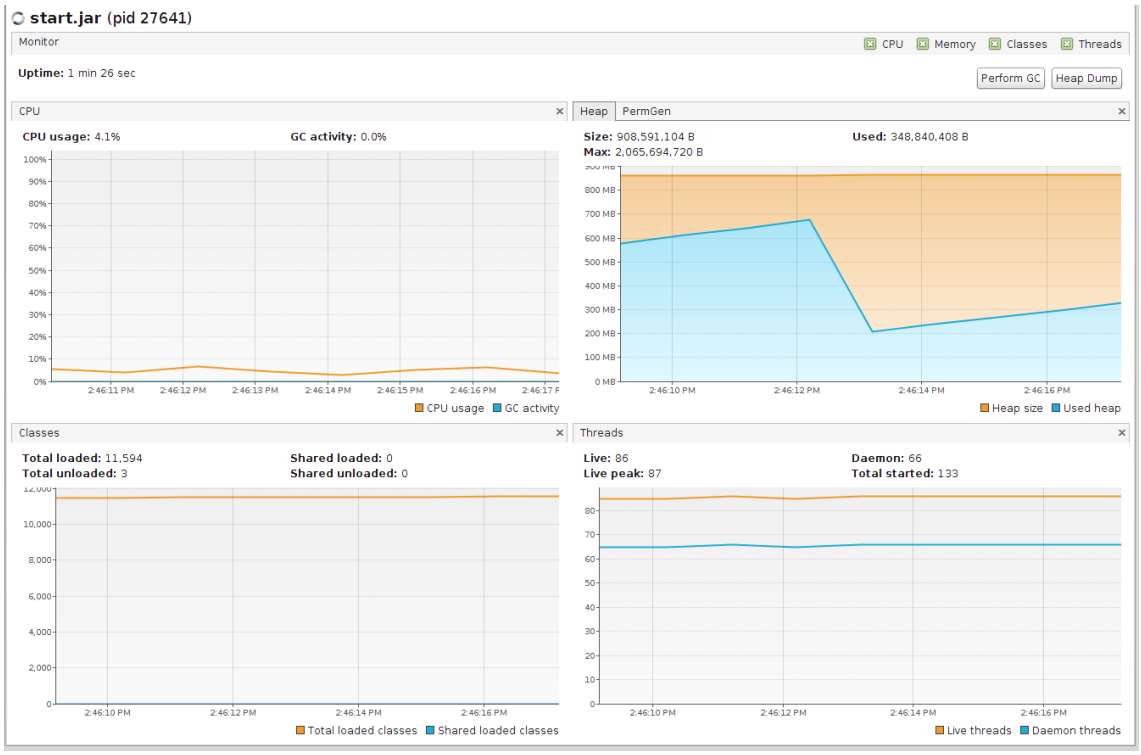

Figure A.143: Resources used when running Exp6 with 60 messages per minute

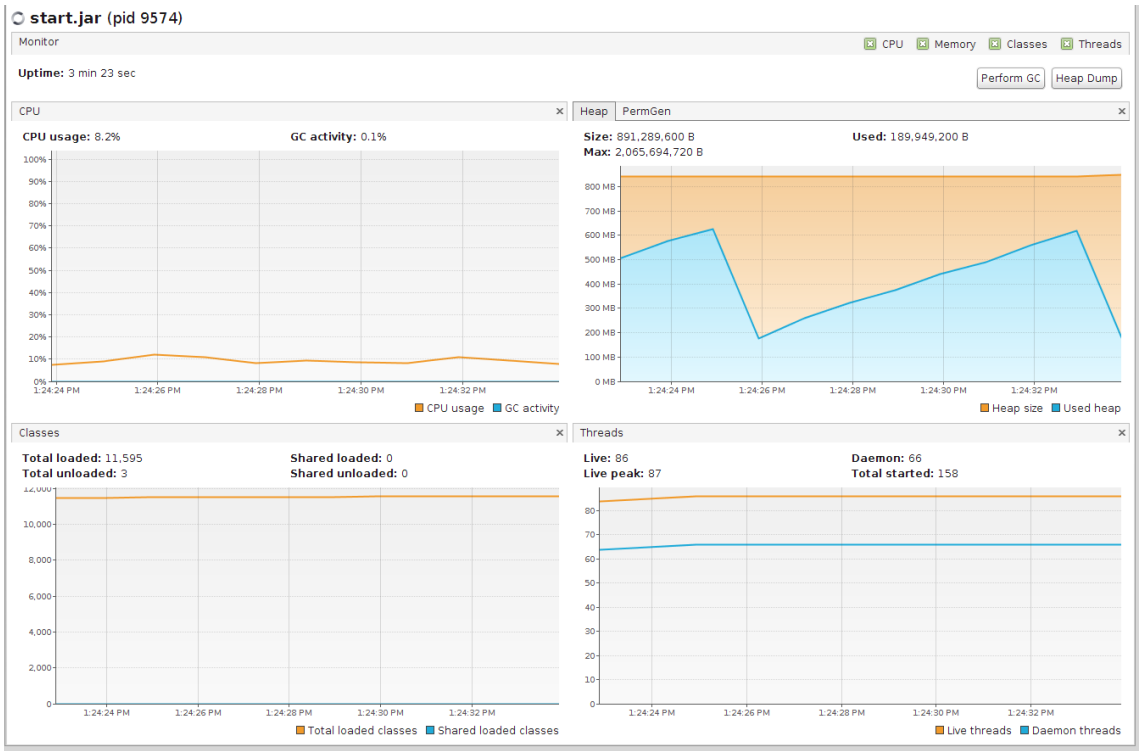

Figure A.144: Resources used when running Exp6 with 120 messages per minute

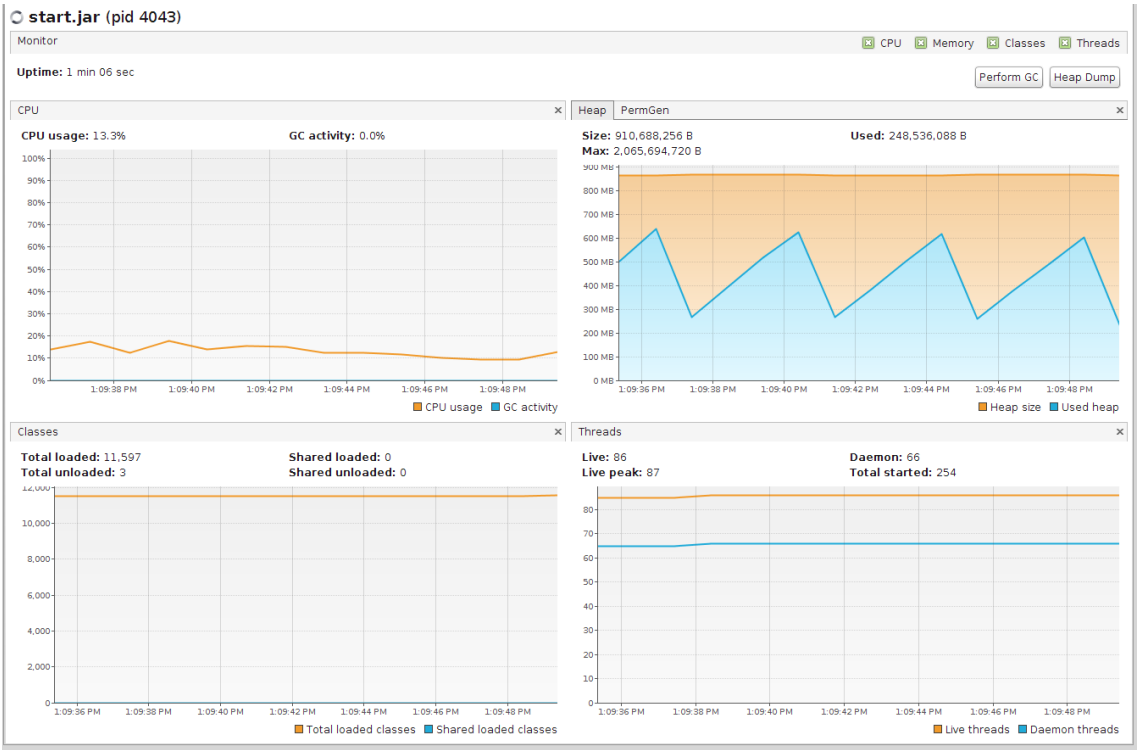

Figure A.145: Resources used when running Exp6 with 240 messages per minute

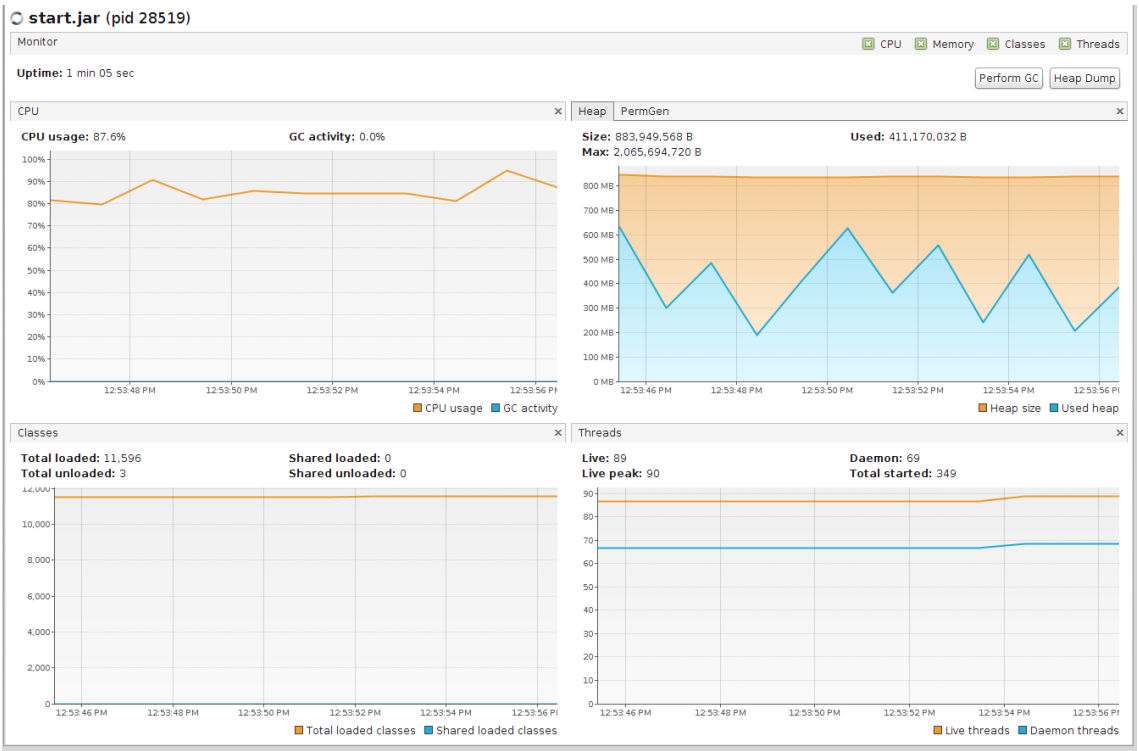

Figure A.146: Resources used when running Exp6 with 480 messages per minute

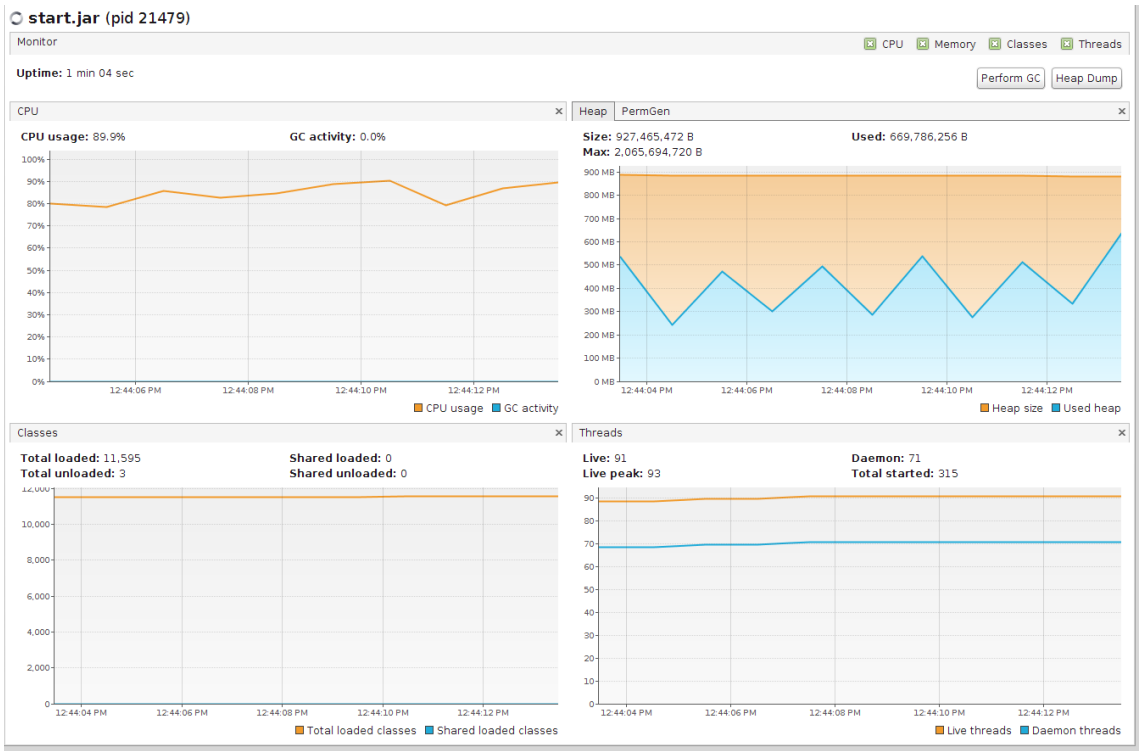

Figure A.147: Resources used when running Exp6 with 1000 messages per minute

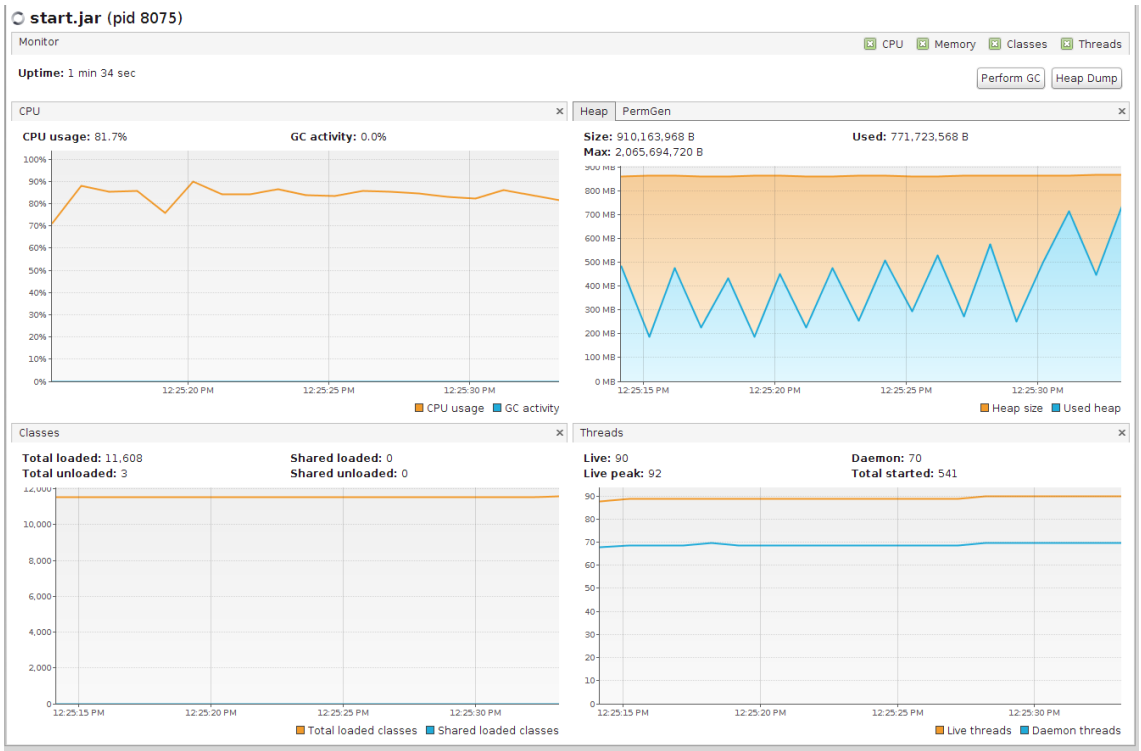

Figure A.148: Resources used when running Exp6 with 2000 messages per minute

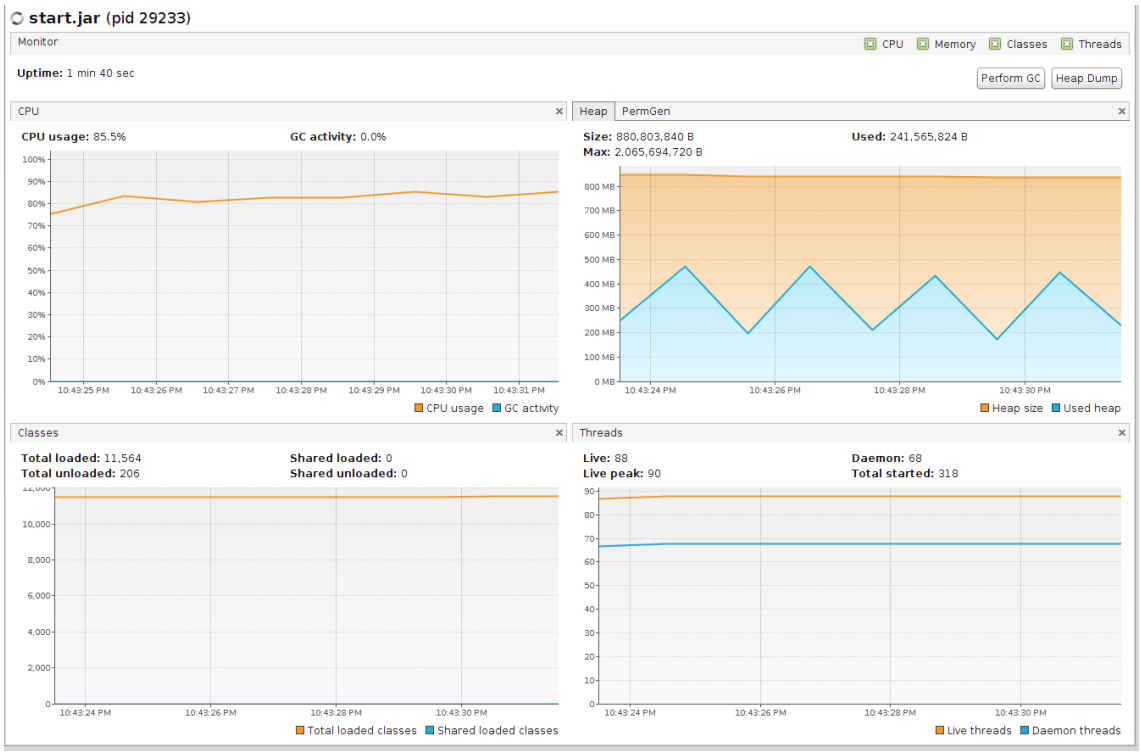

Figure A.149: Resources used when running Exp6 with 4000 messages per minute

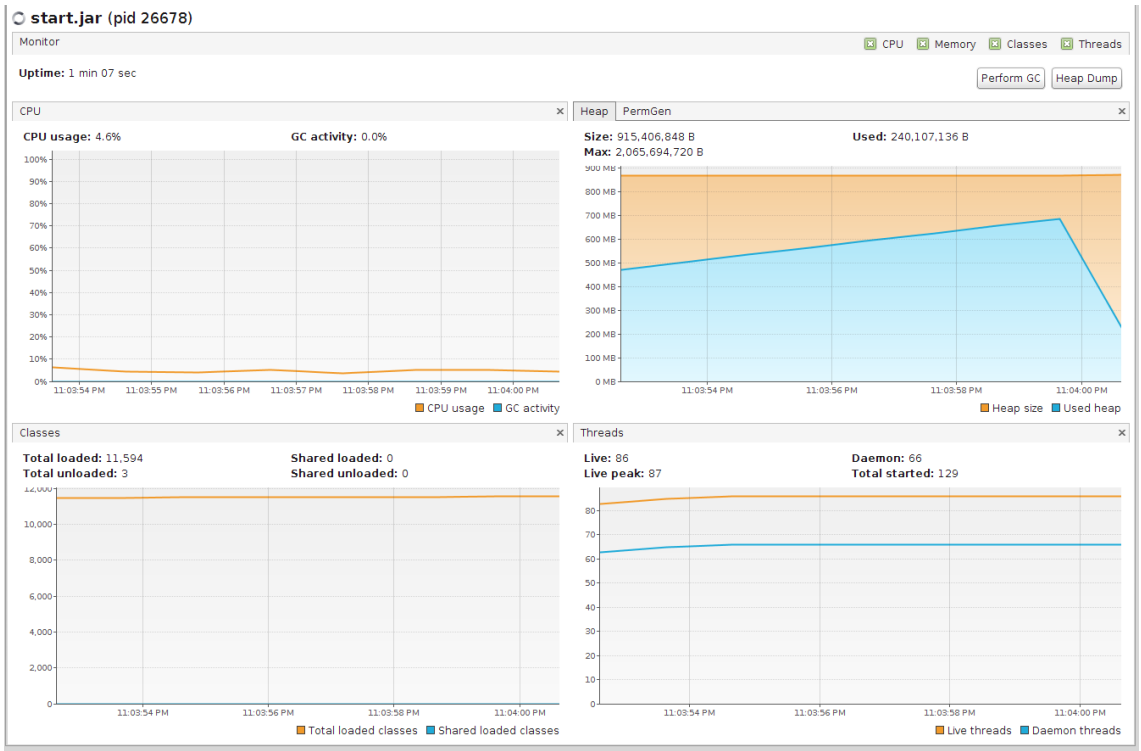

Figure A.150: Resources used when running Exp7 with 60 messages per minute

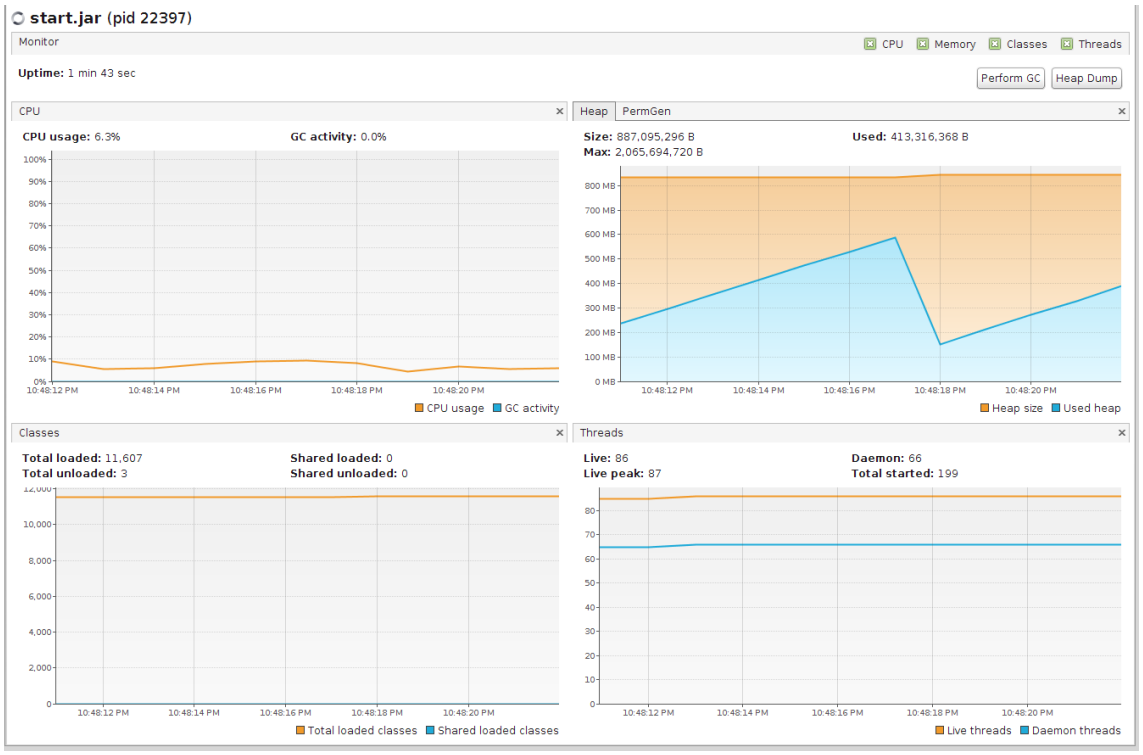

Figure A.151: Resources used when running Exp7 with 120 messages per minute

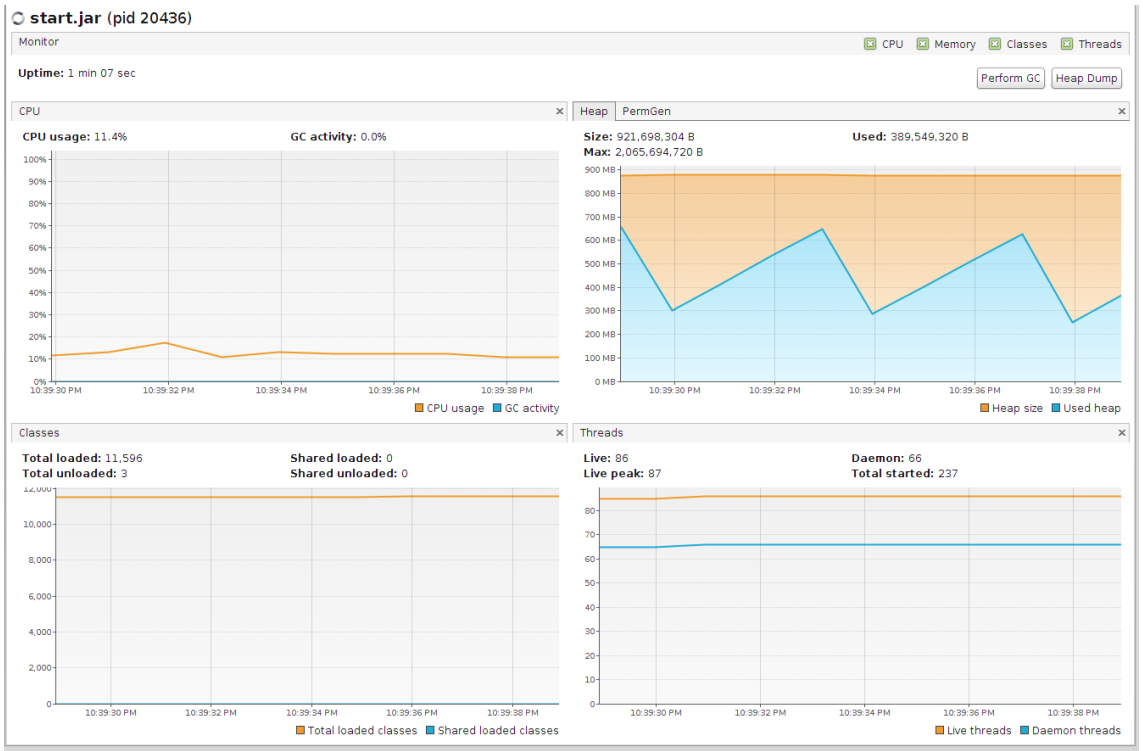

Figure A.152: Resources used when running Exp7 with 240 messages per minute

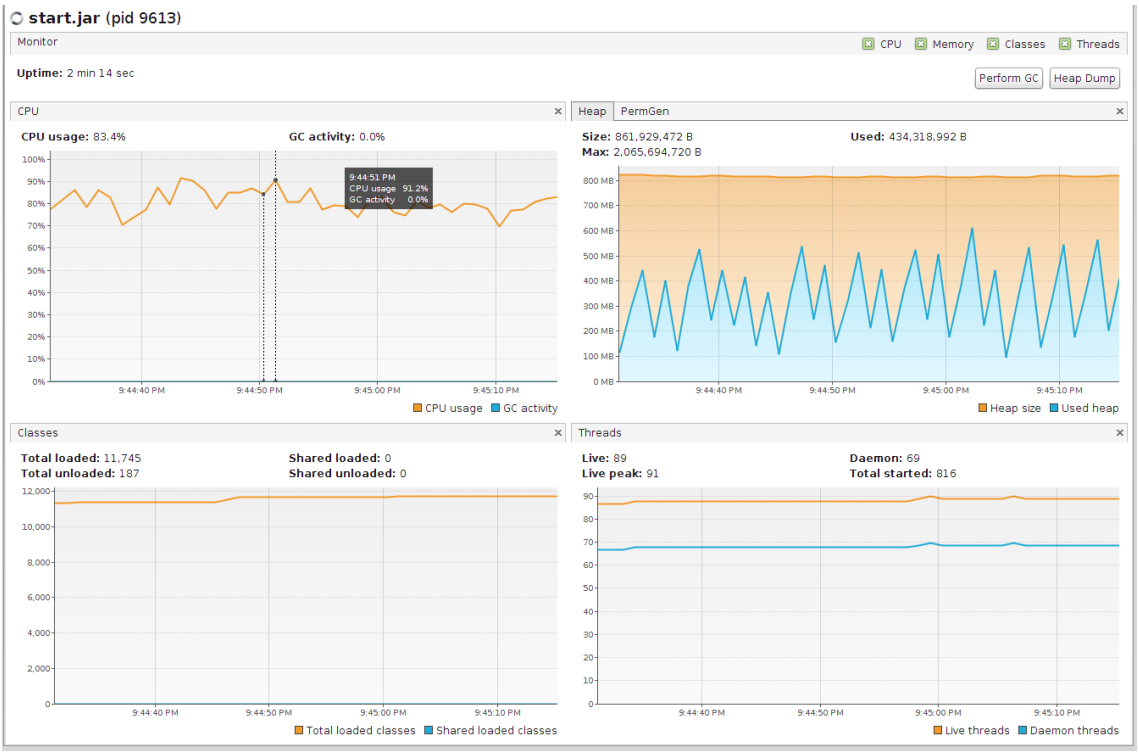

Figure A.153: Resources used when running Exp7 with 480 messages per minute

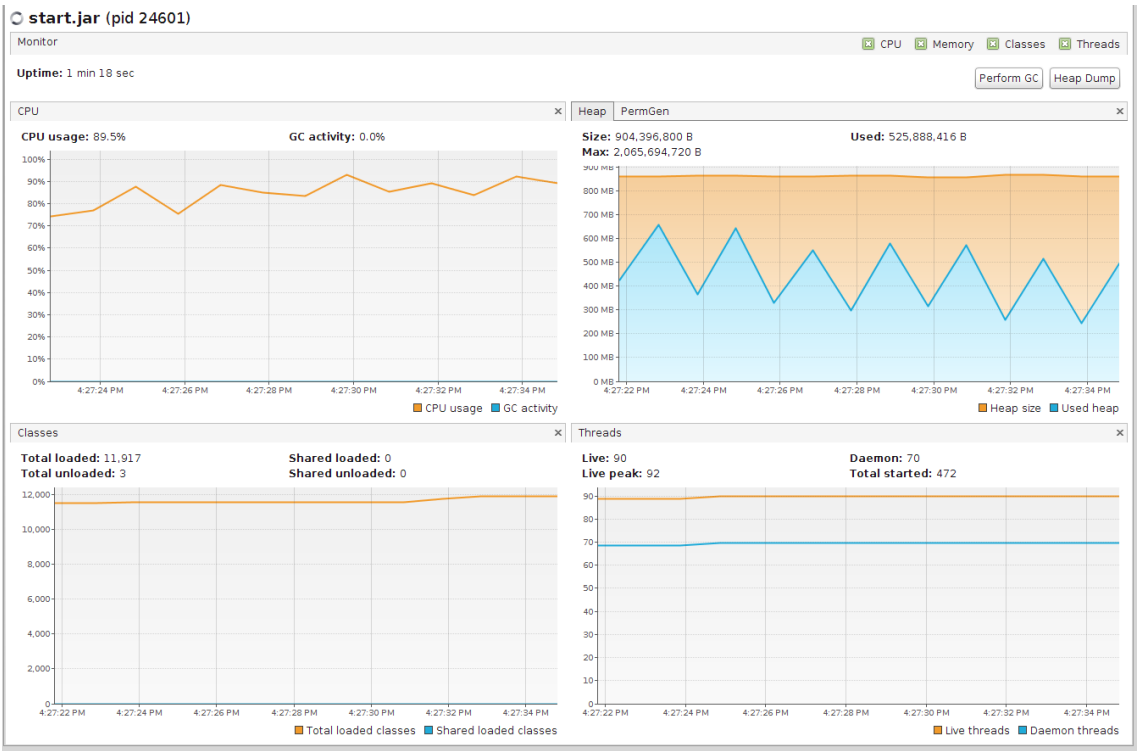

Figure A.154: Resources used when running Exp7 with 1000 messages per minute

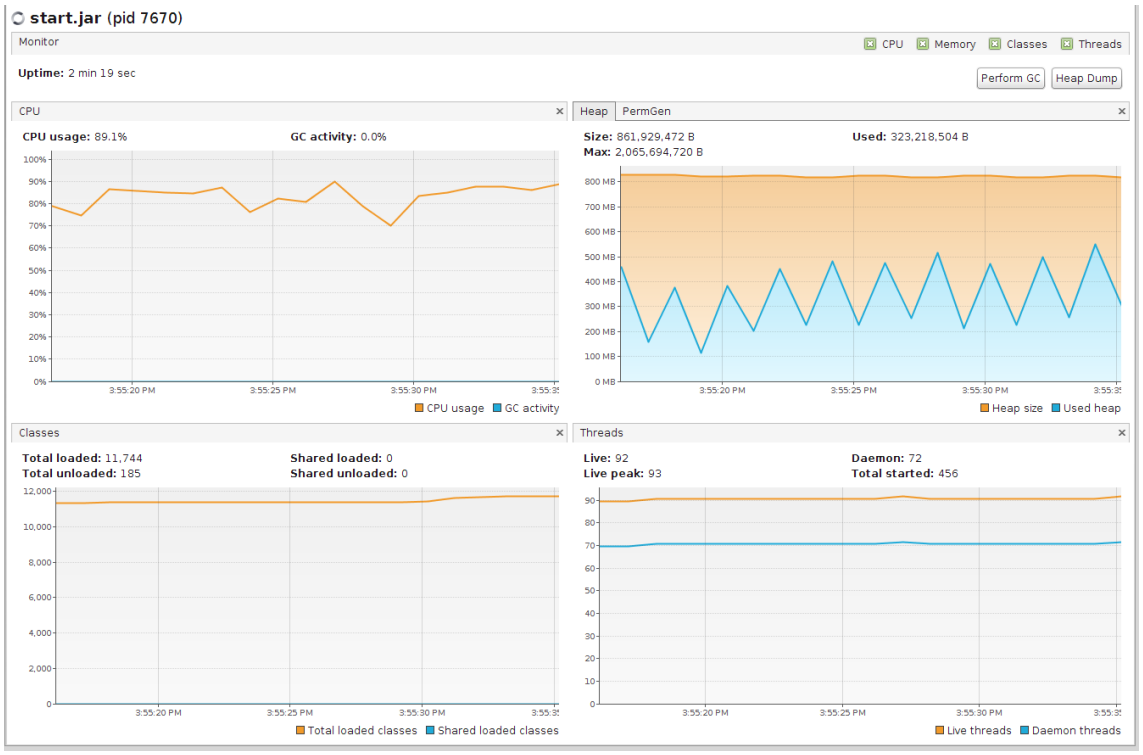

Figure A.155: Resources used when running Exp7 with 2000 messages per minute

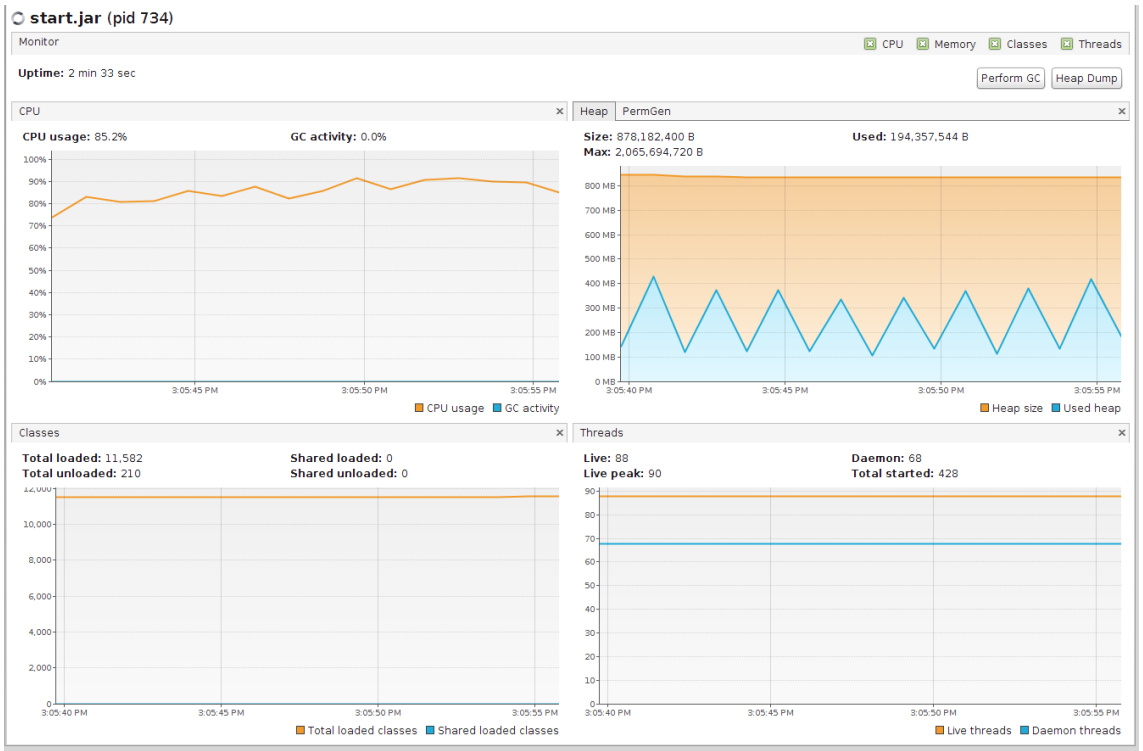

Figure A.156: Resources used when running Exp7 with 4000 messages per minute

A.5.2 Comparision graphs of middleware delay

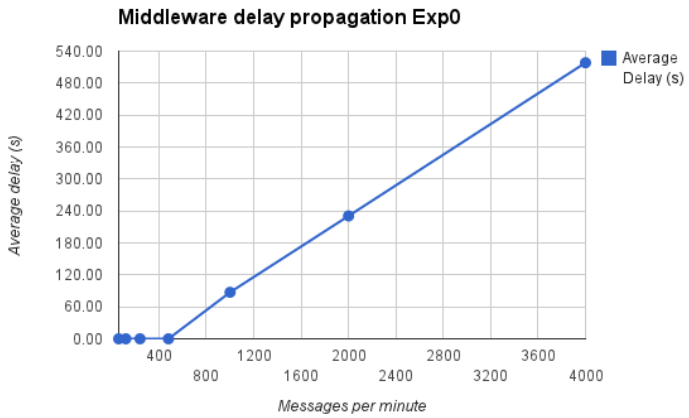

Figure A.157: Middleware delay propagation using Exp0

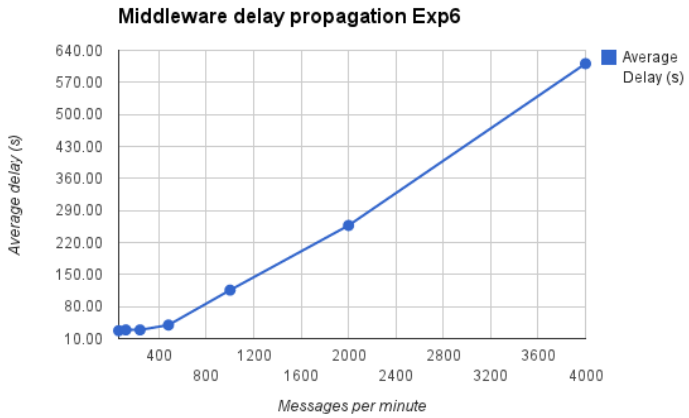

Figure A.158: Middleware delay propagation using Exp6

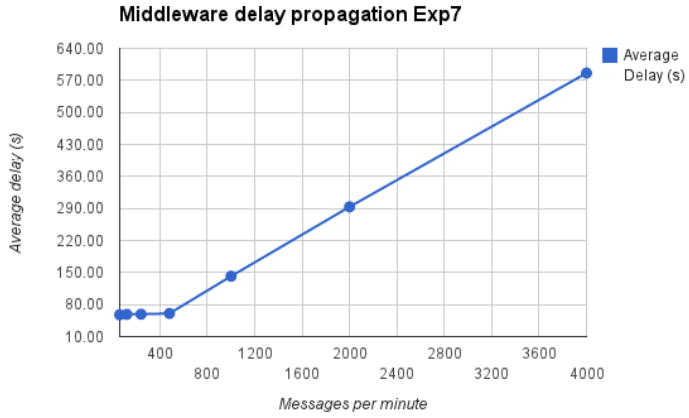

Figure A.159: Middleware delay propagation using Exp7

A.6 1 source, 4 sessions, 1 client, SEDA full info

A.6.1 visualVM monitor screenshots

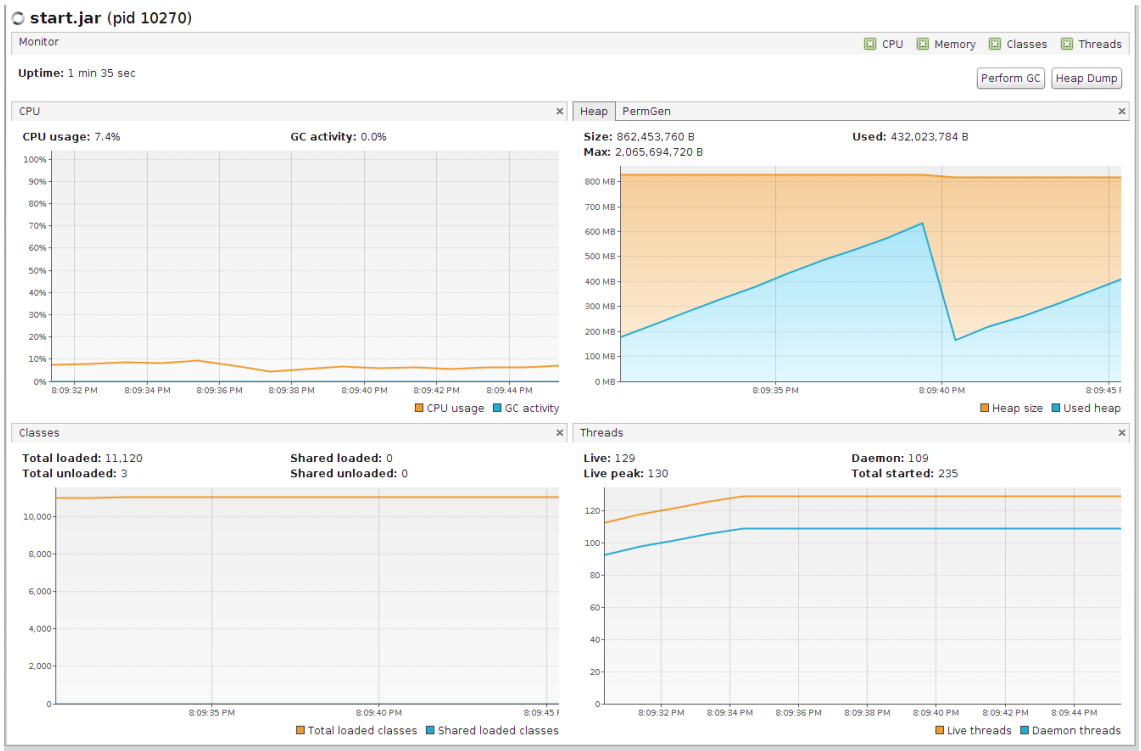

Figure A.160: Resources used when running Exp0 with 60 messages per minute

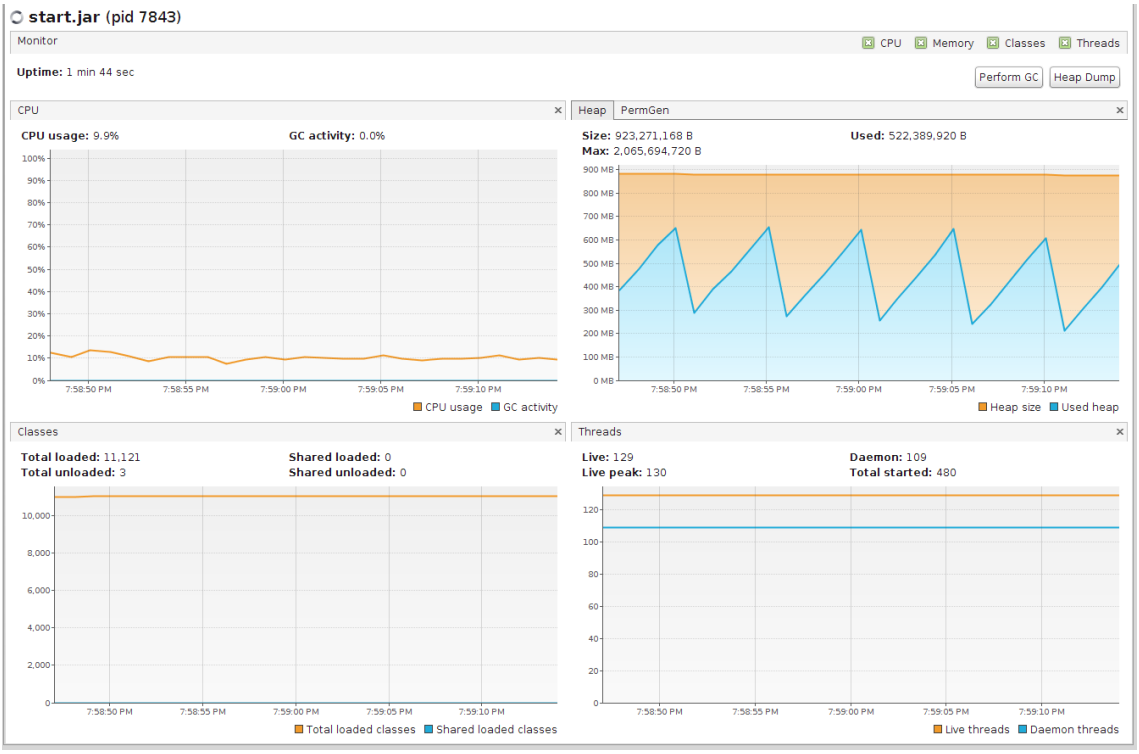

Figure A.161: Resources used when running Exp0 with 120 messages per minute

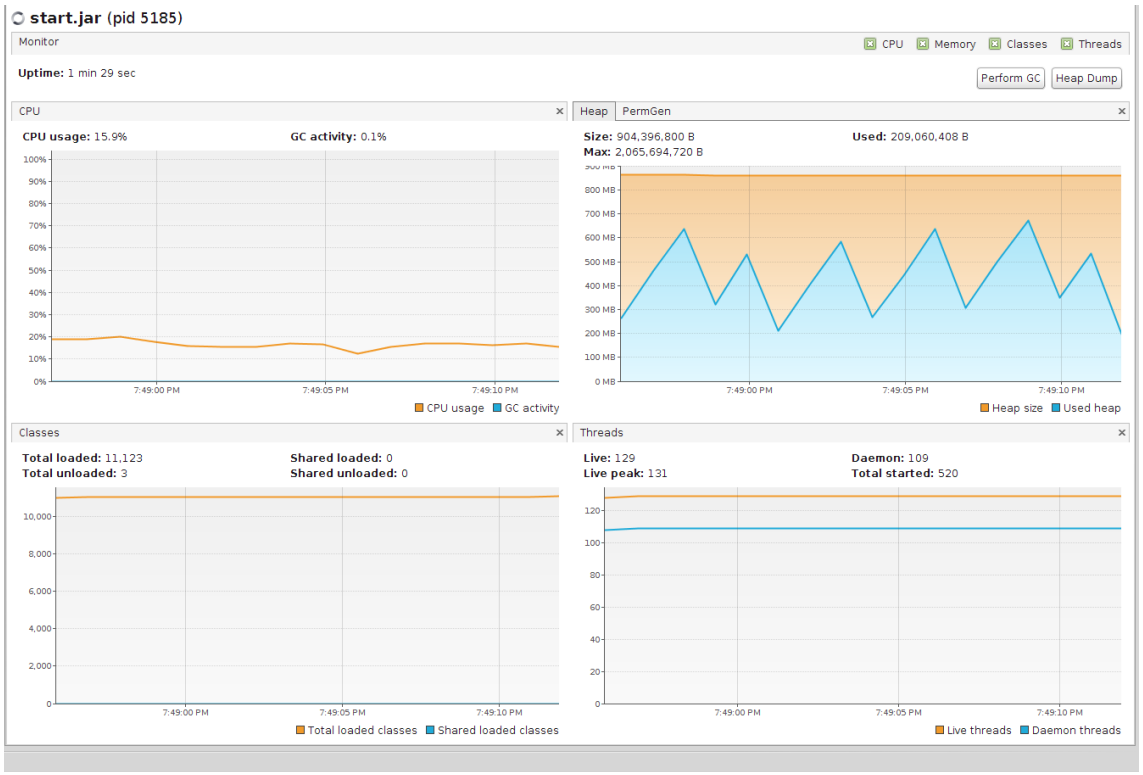

Figure A.162: Resources used when running Exp0 with 240 messages per minute

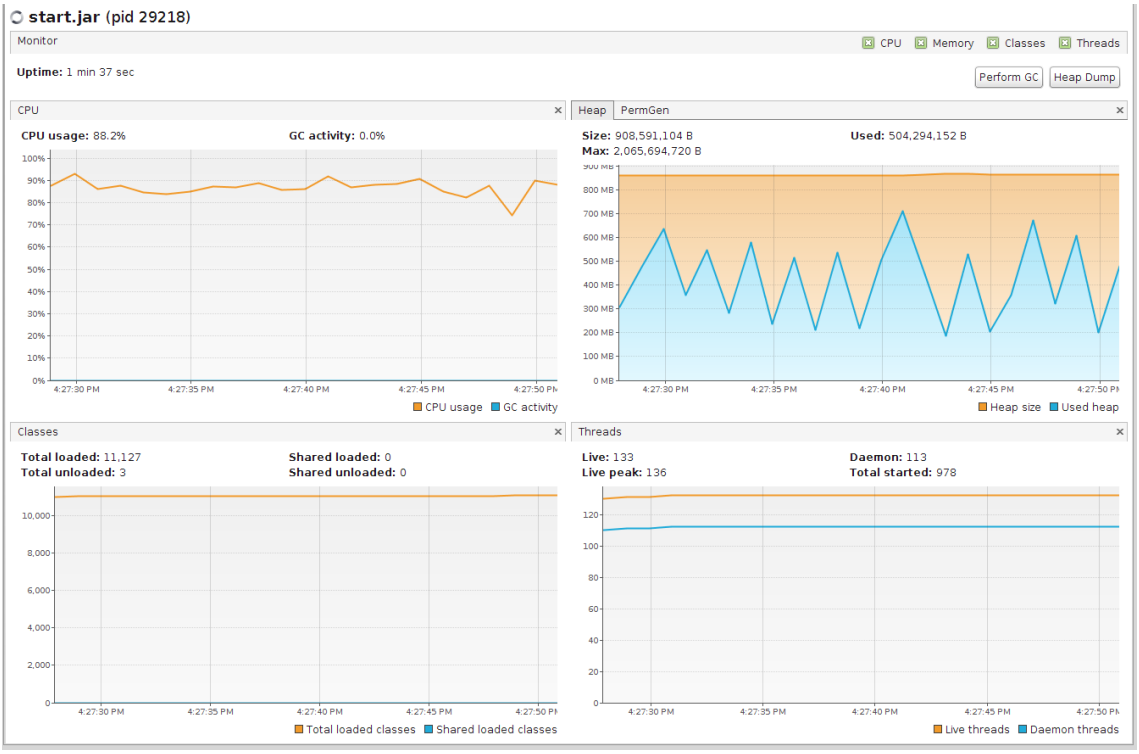

Figure A.163: Resources used when running Exp0 with 480 messages per minute

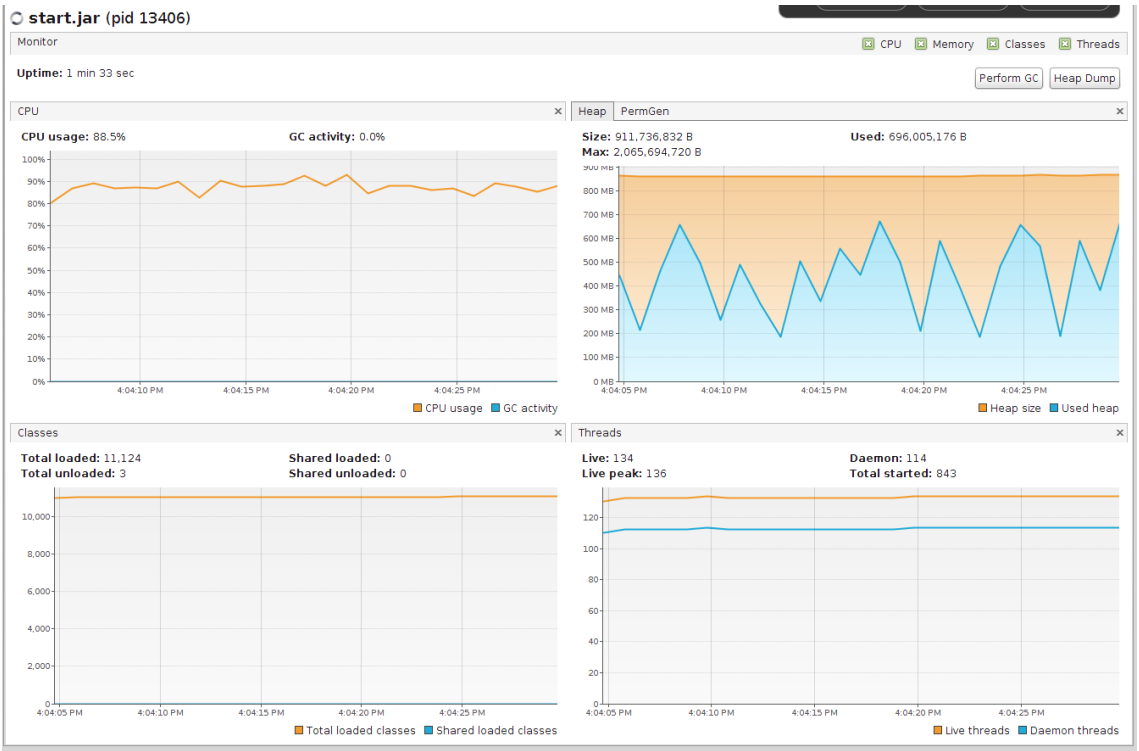

Figure A.164: Resources used when running Exp0 with 1000 messages per minute

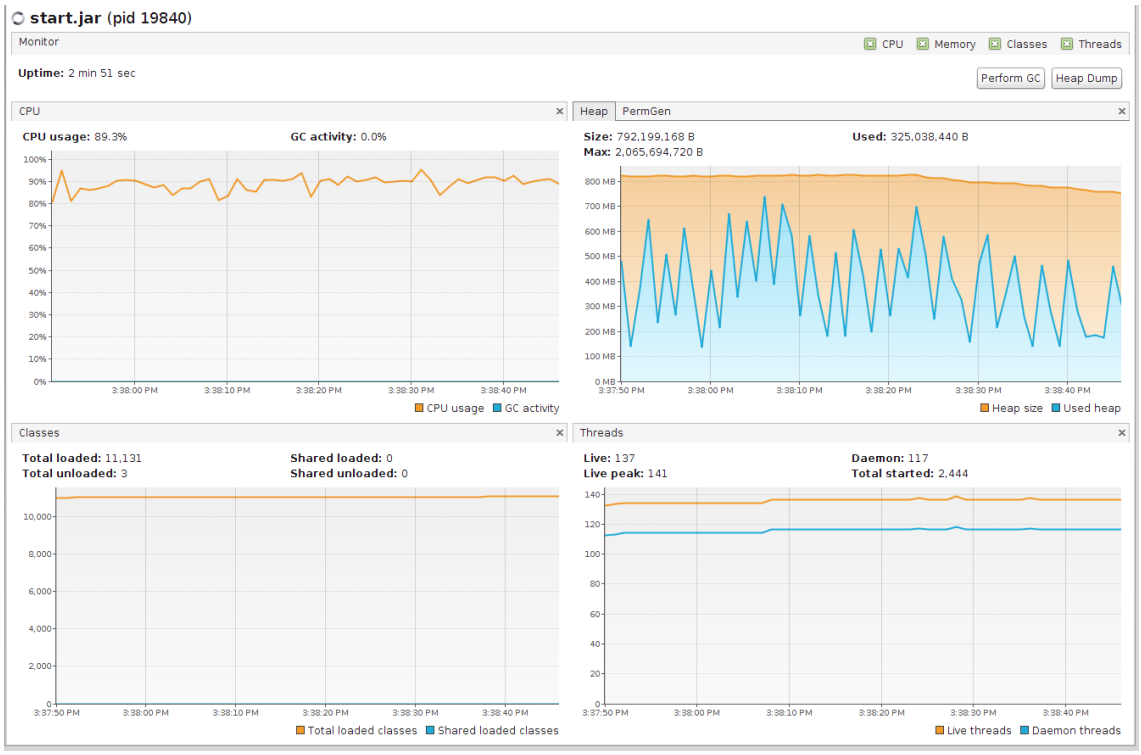

Figure A.165: Resources used when running Exp0 with 2000 messages per minute

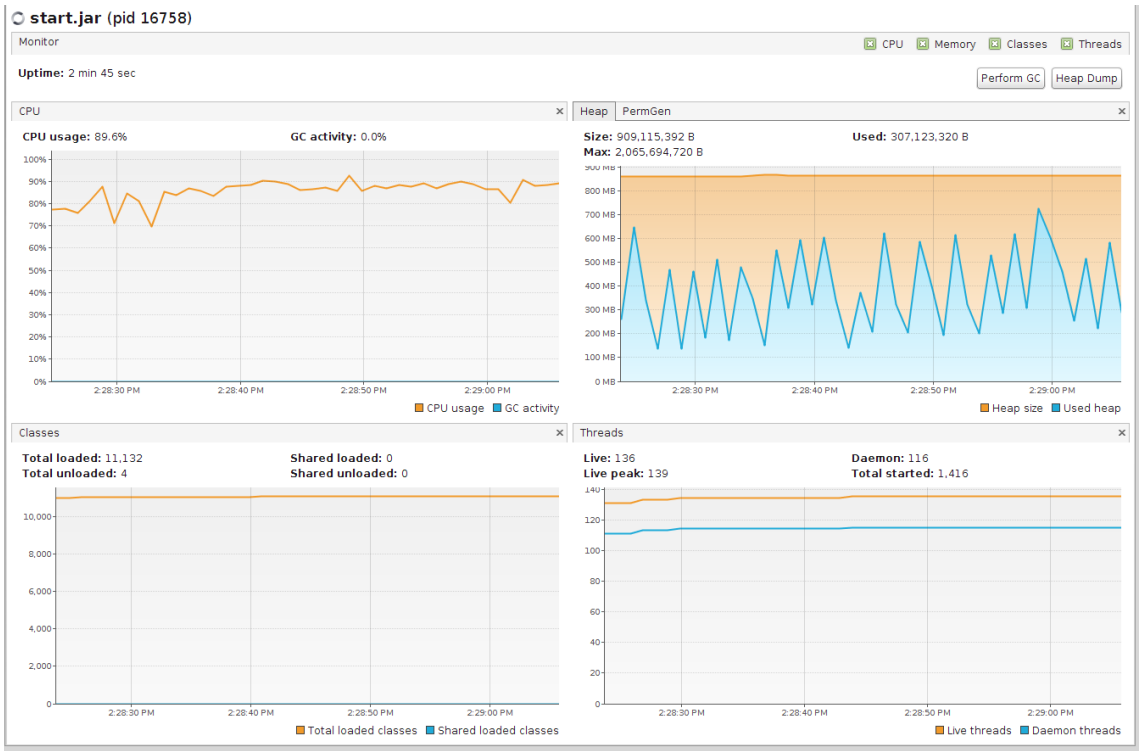

Figure A.166: Resources used when running Exp0 with 4000 messages per minute

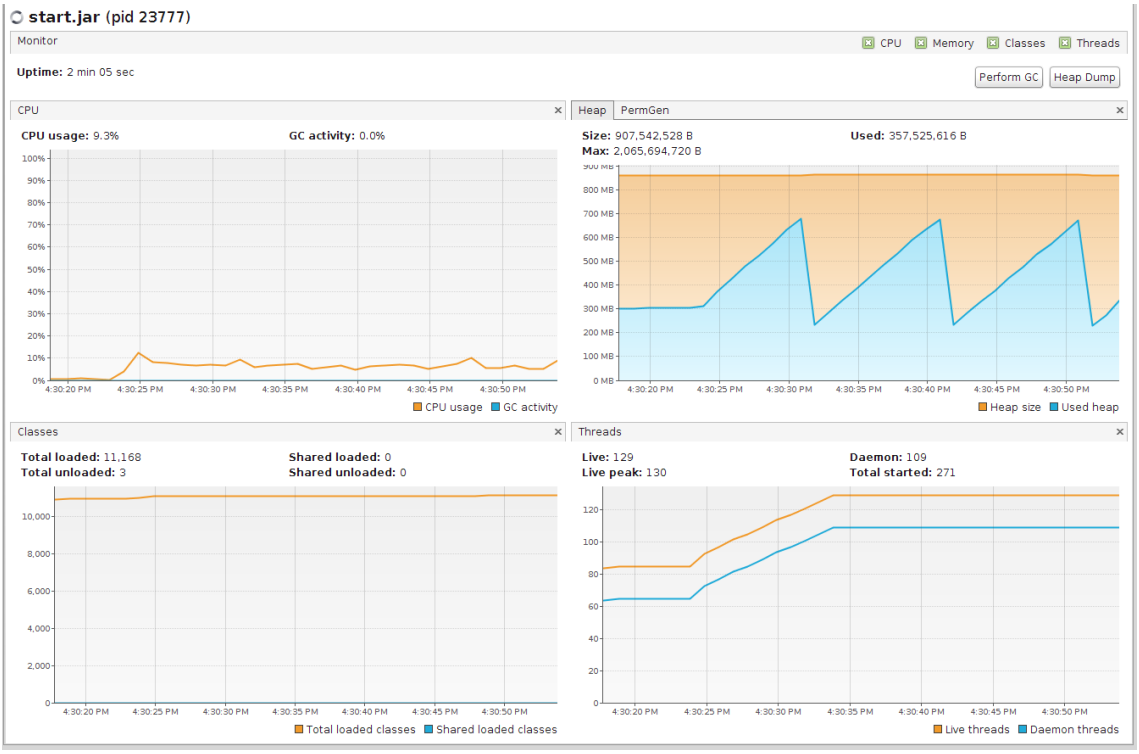

Figure A.167: Resources used when running Exp6 with 60 messages per minute

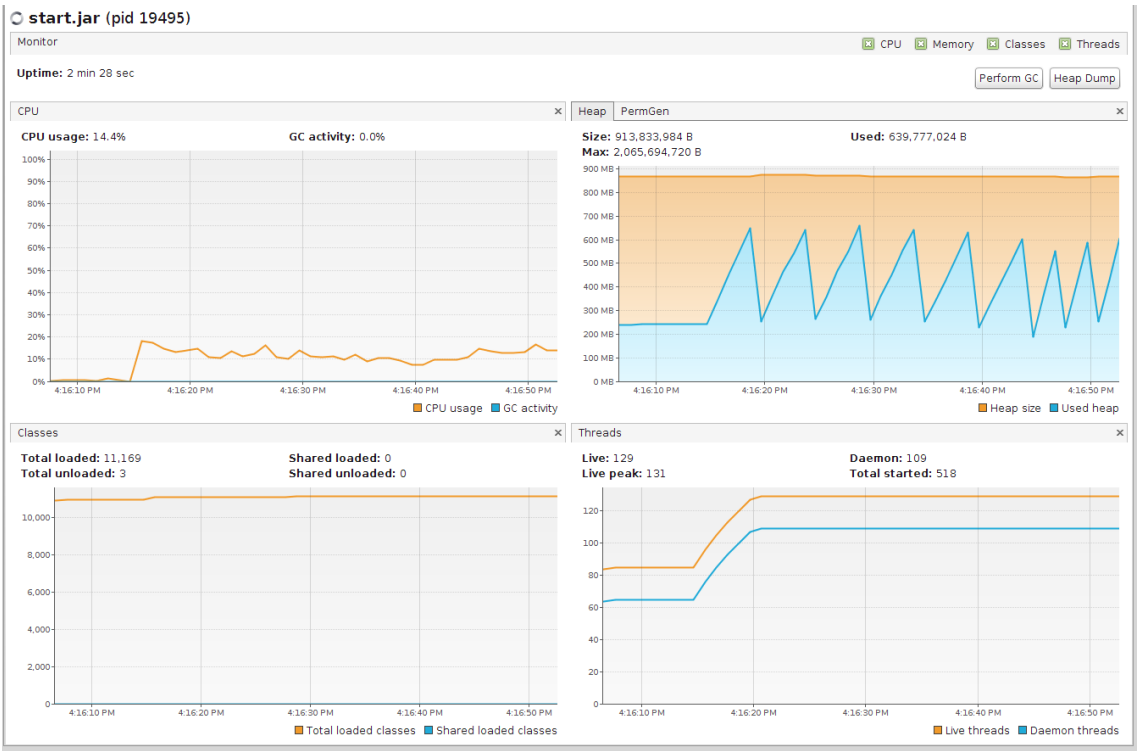

Figure A.168: Resources used when running Exp6 with 120 messages per minute

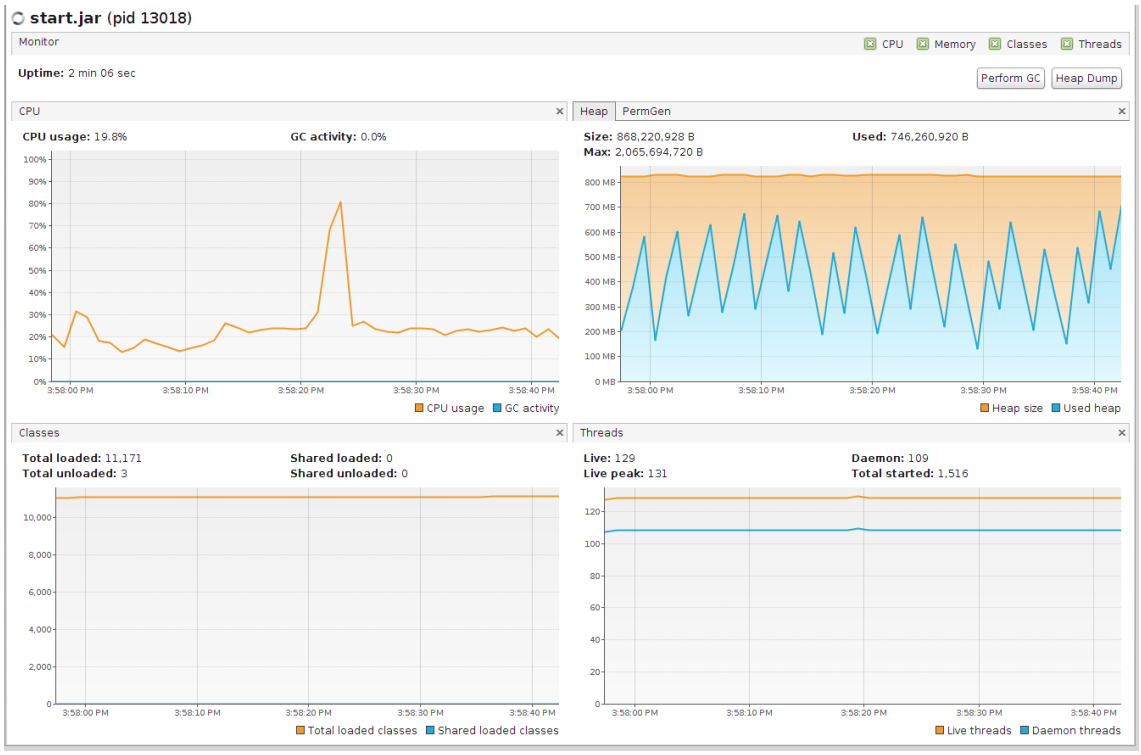

Figure A.169: Resources used when running Exp6 with 240 messages per minute

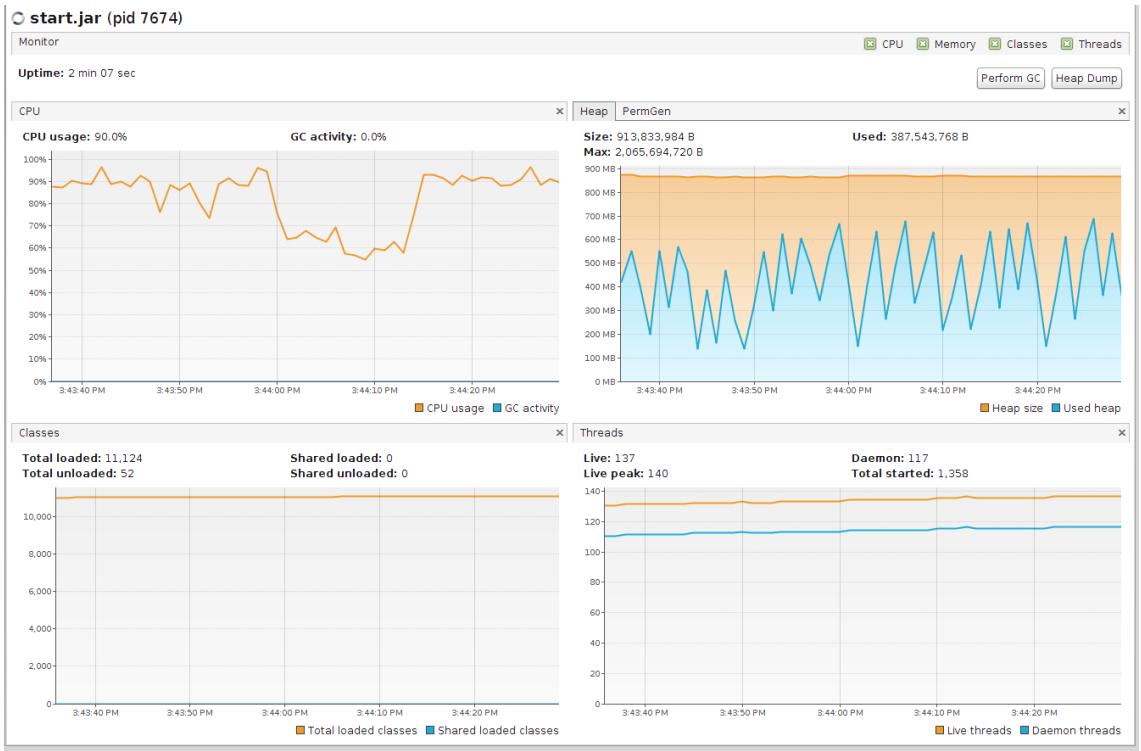

Figure A.170: Resources used when running Exp6 with 480 messages per minute

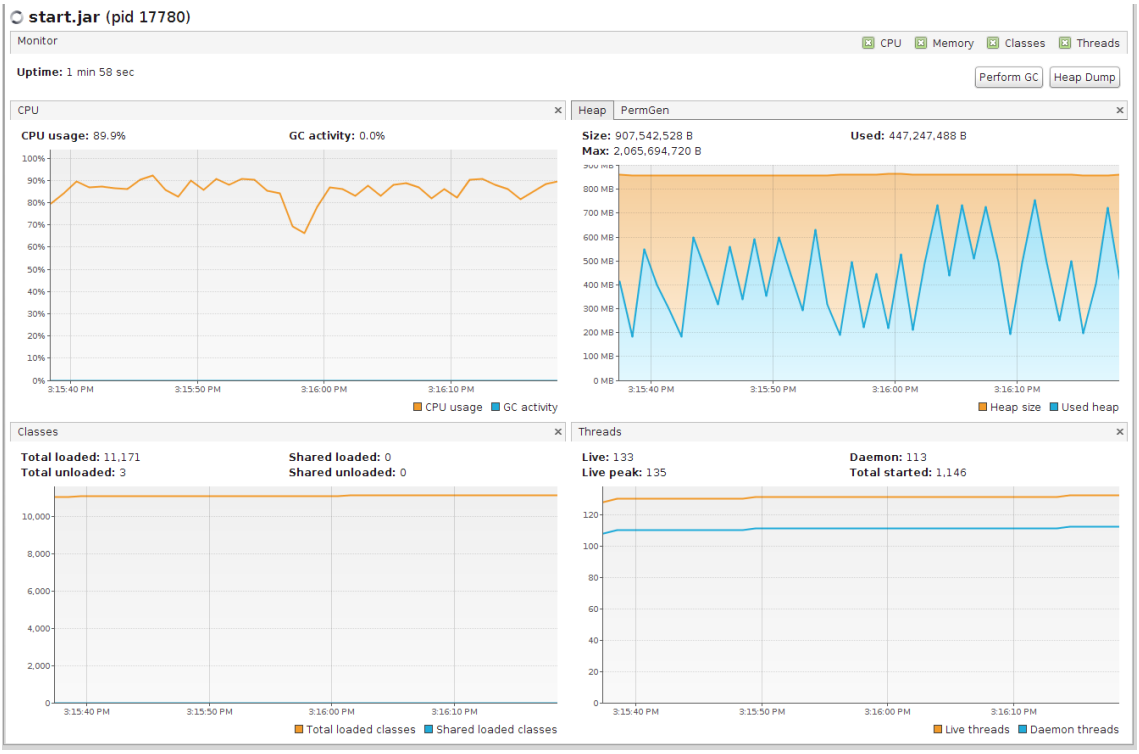

Figure A.171: Resources used when running Exp6 with 1000 messages per minute

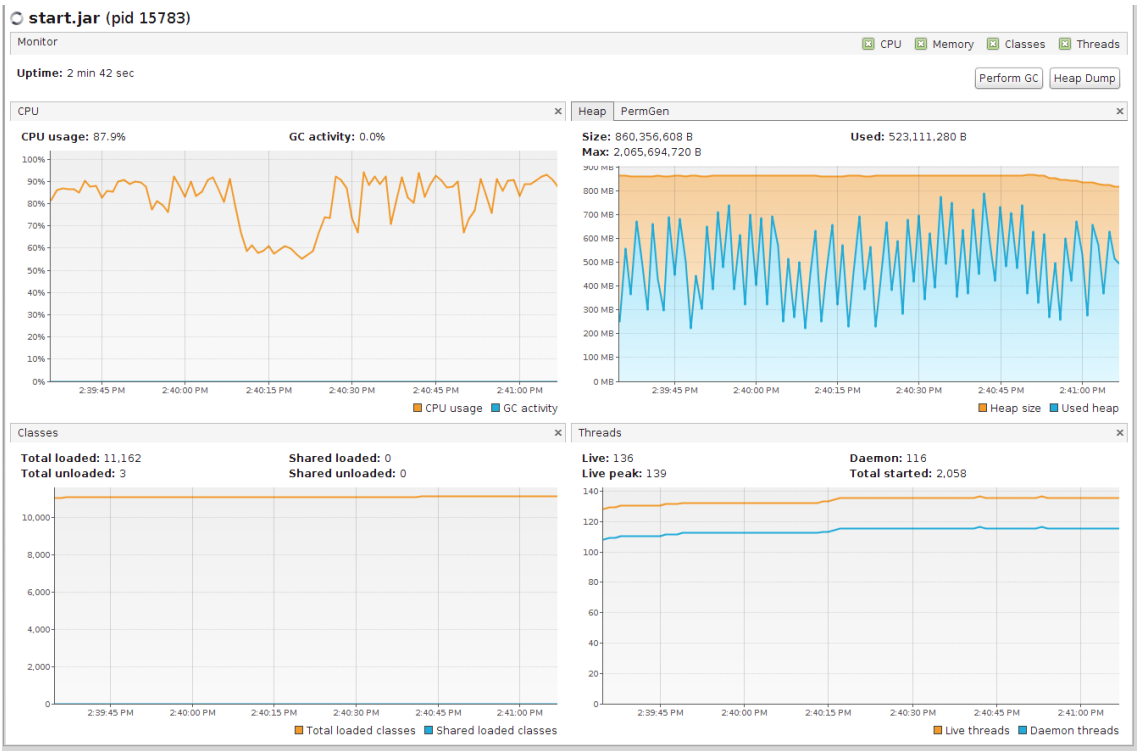

Figure A.172: Resources used when running Exp6 with 2000 messages per minute

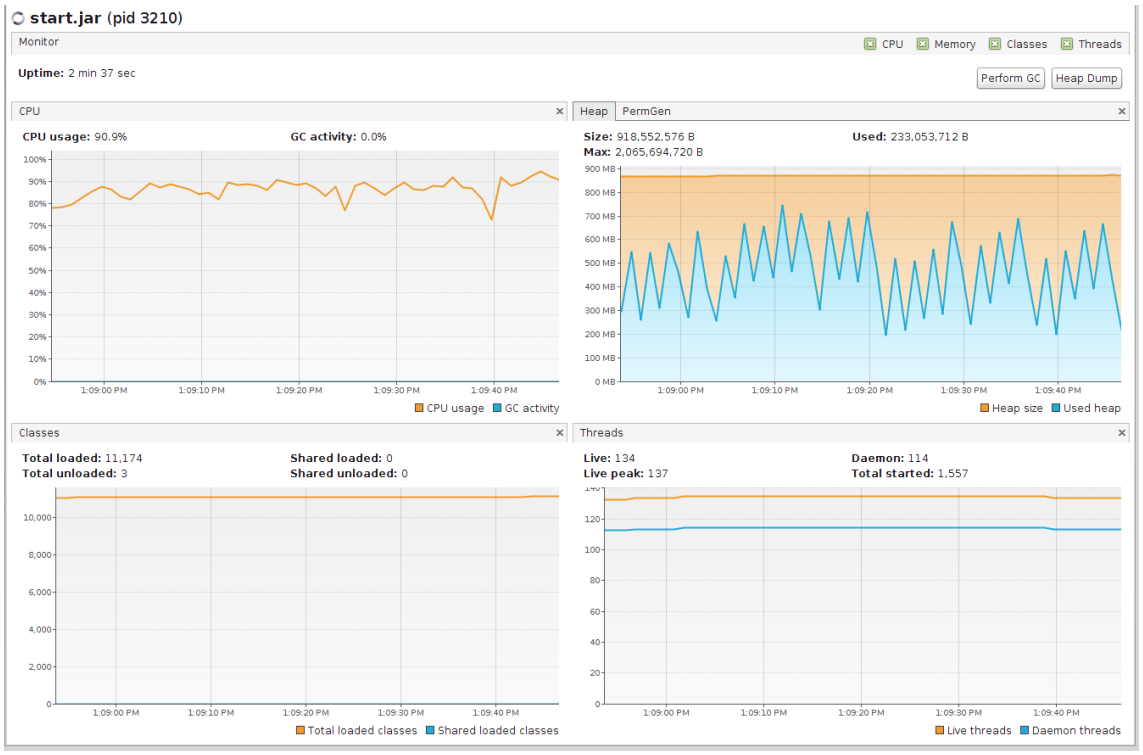

Figure A.173: Resources used when running Exp6 with 4000 messages per minute

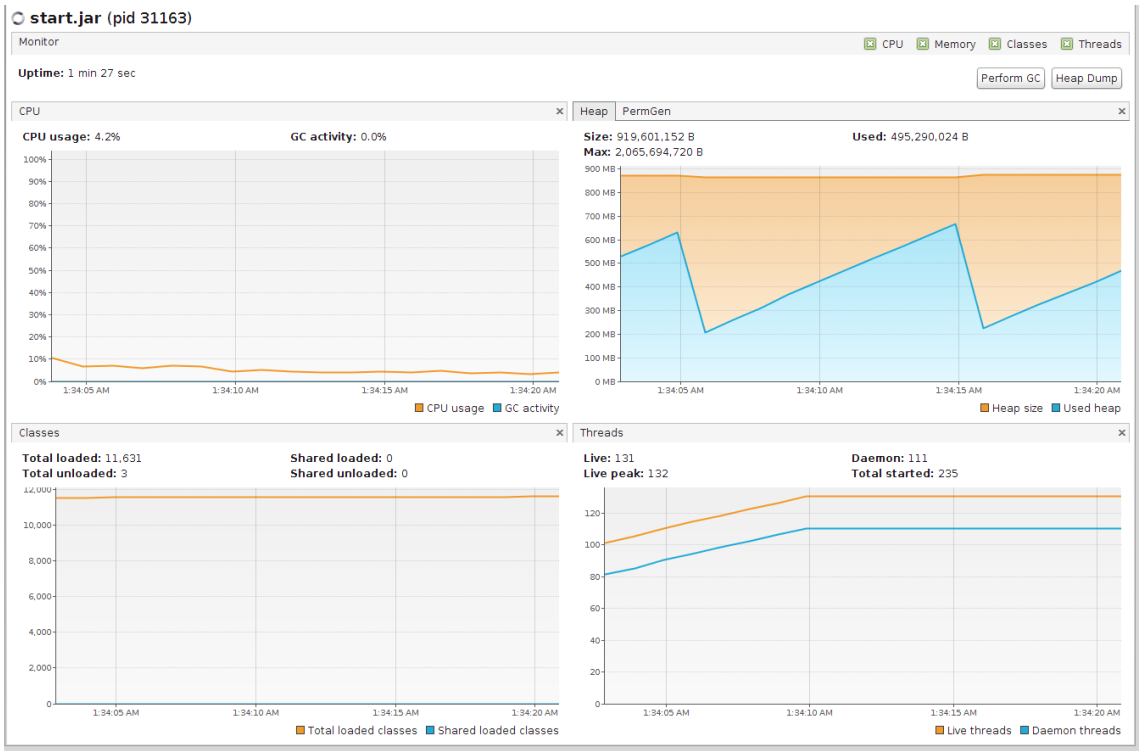

Figure A.174: Resources used when running Exp7 with 60 messages per minute

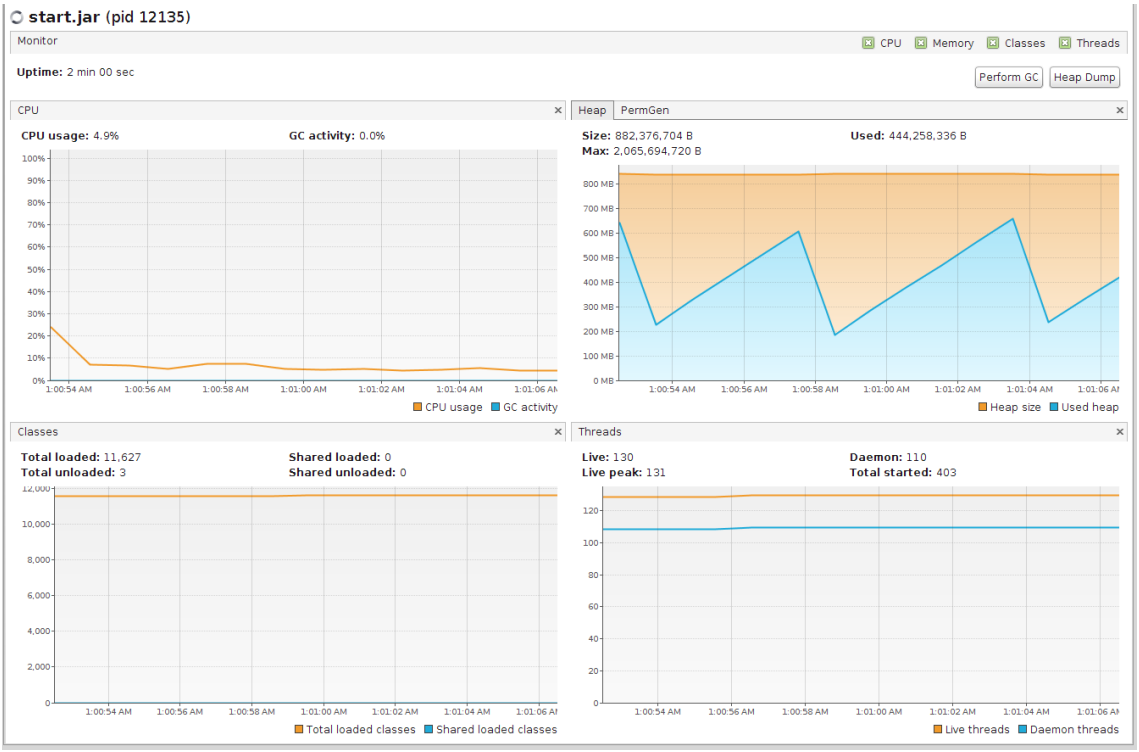

Figure A.175: Resources used when running Exp7 with 120 messages per minute

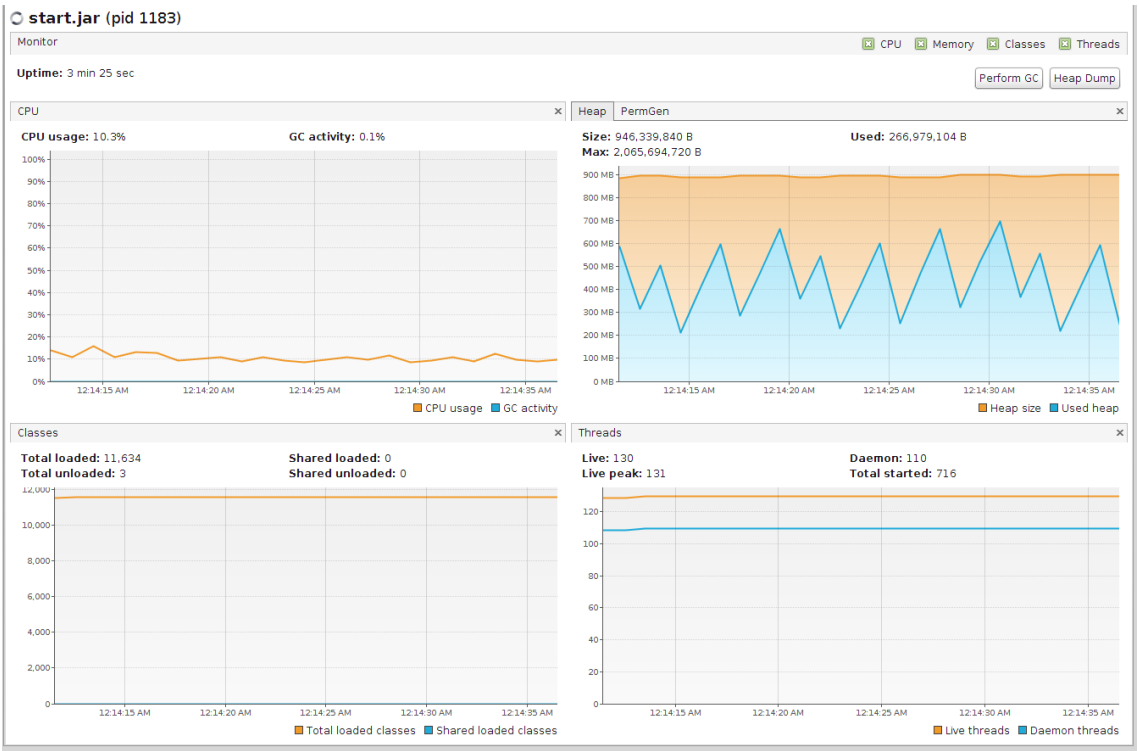

Figure A.176: Resources used when running Exp7 with 240 messages per minute

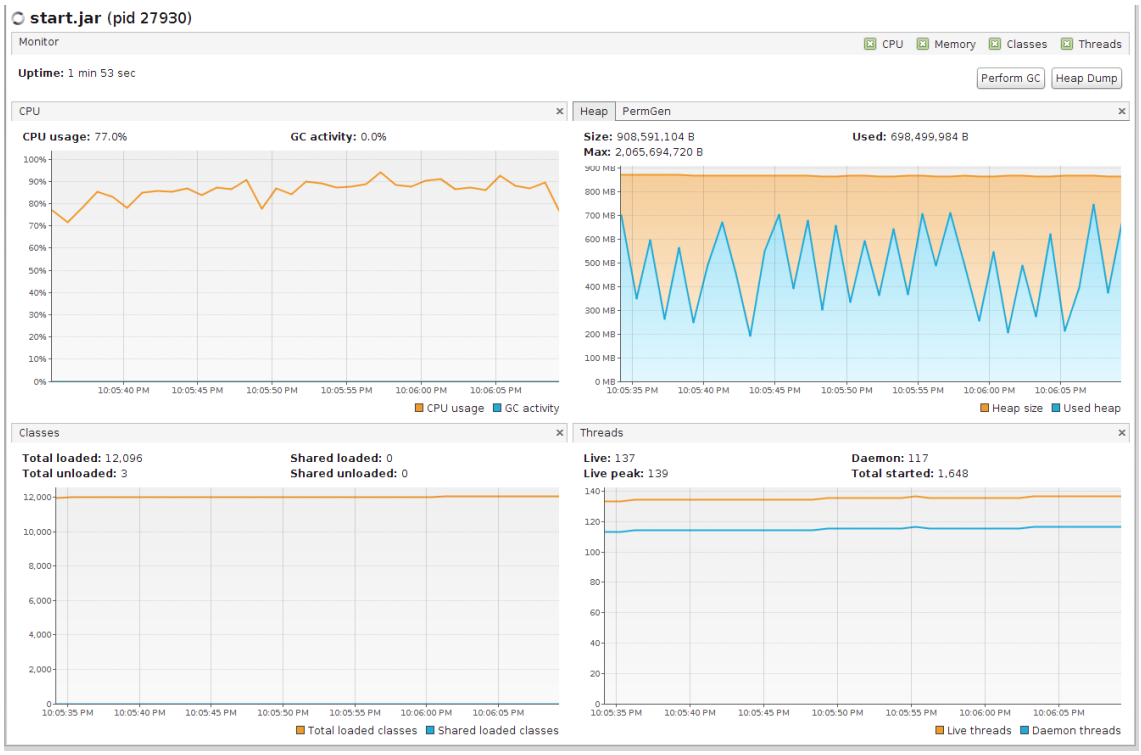

Figure A.177: Resources used when running Exp7 with 480 messages per minute

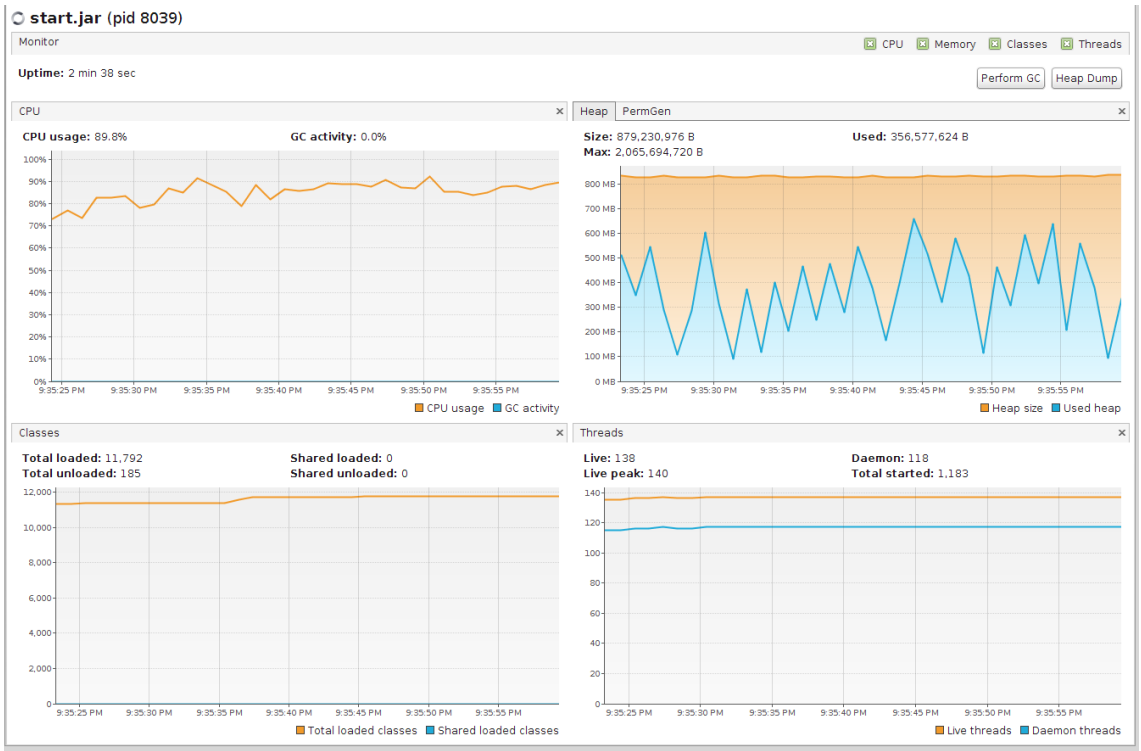

Figure A.178: Resources used when running Exp7 with 1000 messages per minute

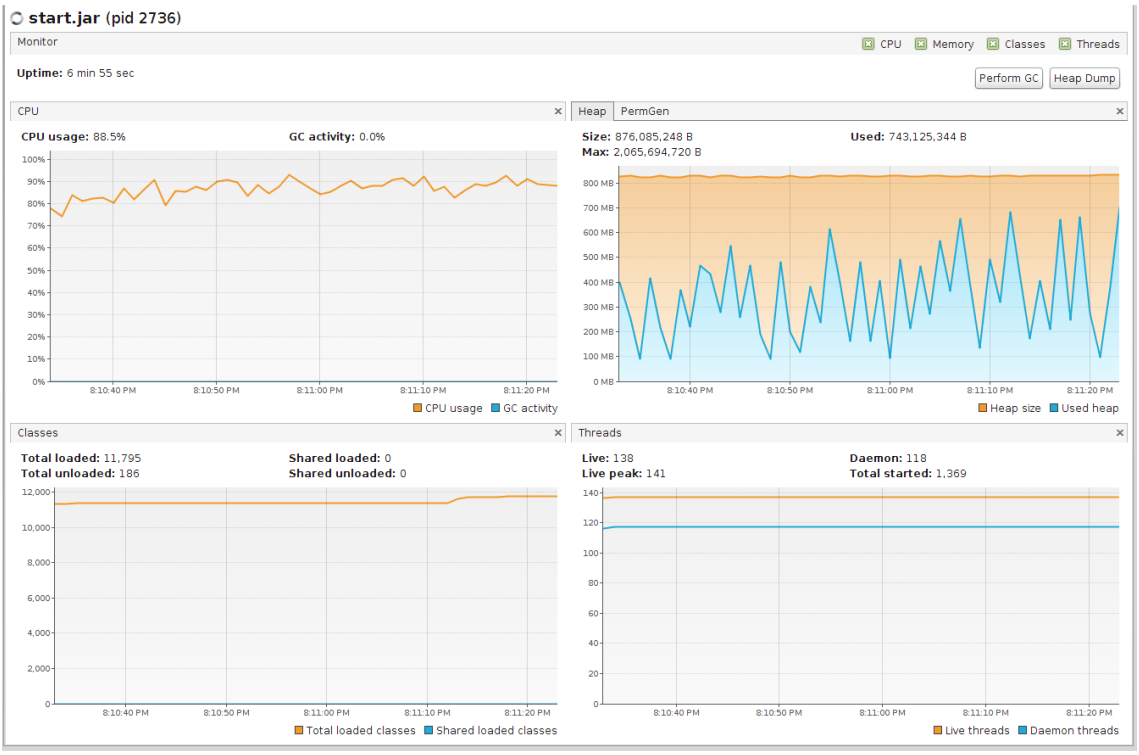

Figure A.179: Resources used when running Exp7 with 2000 messages per minute

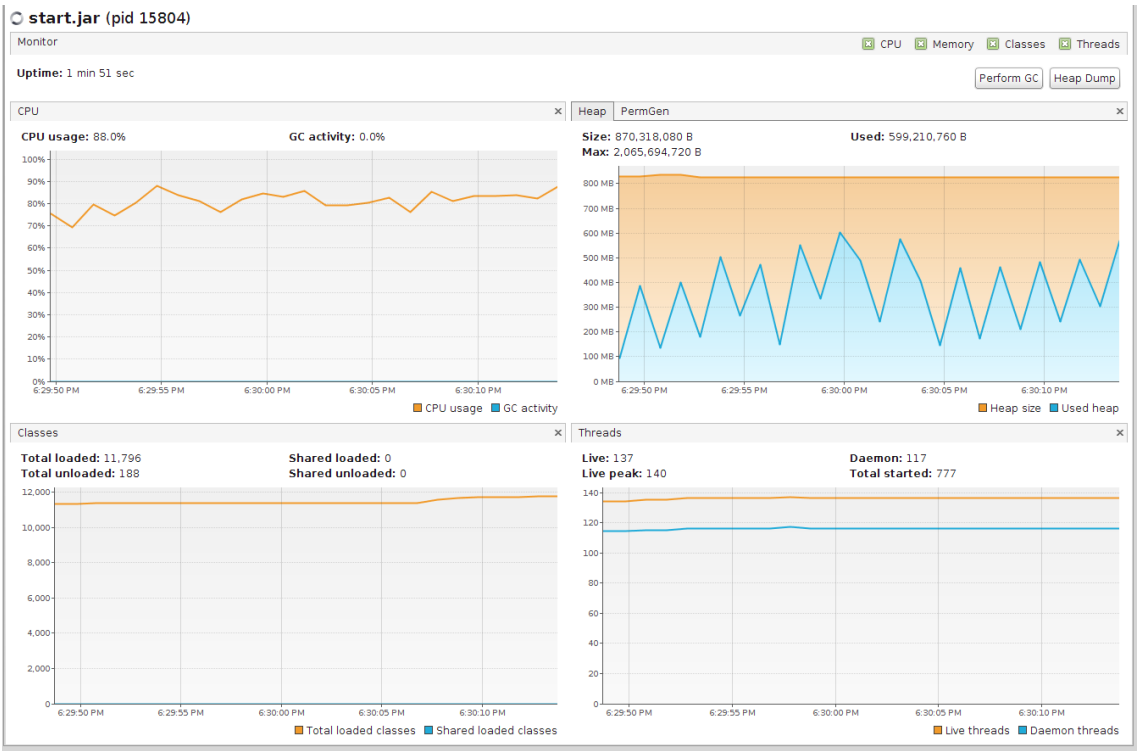

Figure A.180: Resources used when running Exp7 with 4000 messages per minute

A.6.2 Comparision graphs of middleware delay

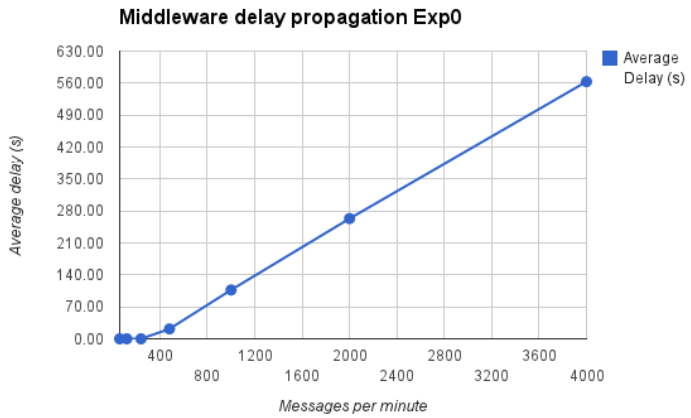

Figure A.181: Middleware delay propagation using Exp0

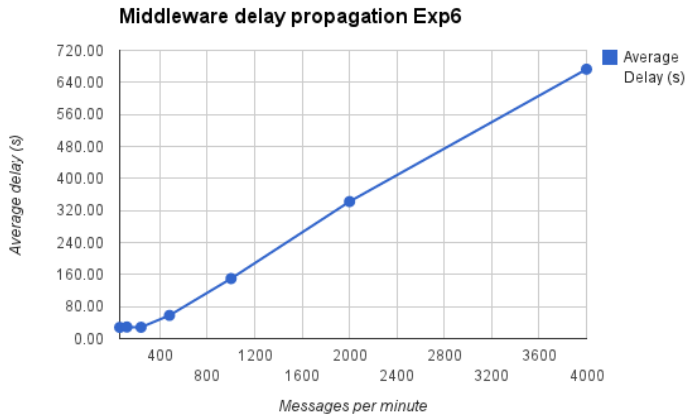

Figure A.182: Middleware delay propagation using Exp6

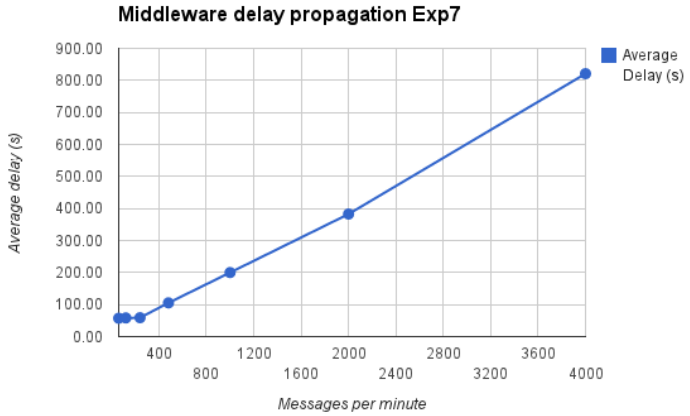

Figure A.183: Middleware delay propagation using Exp7